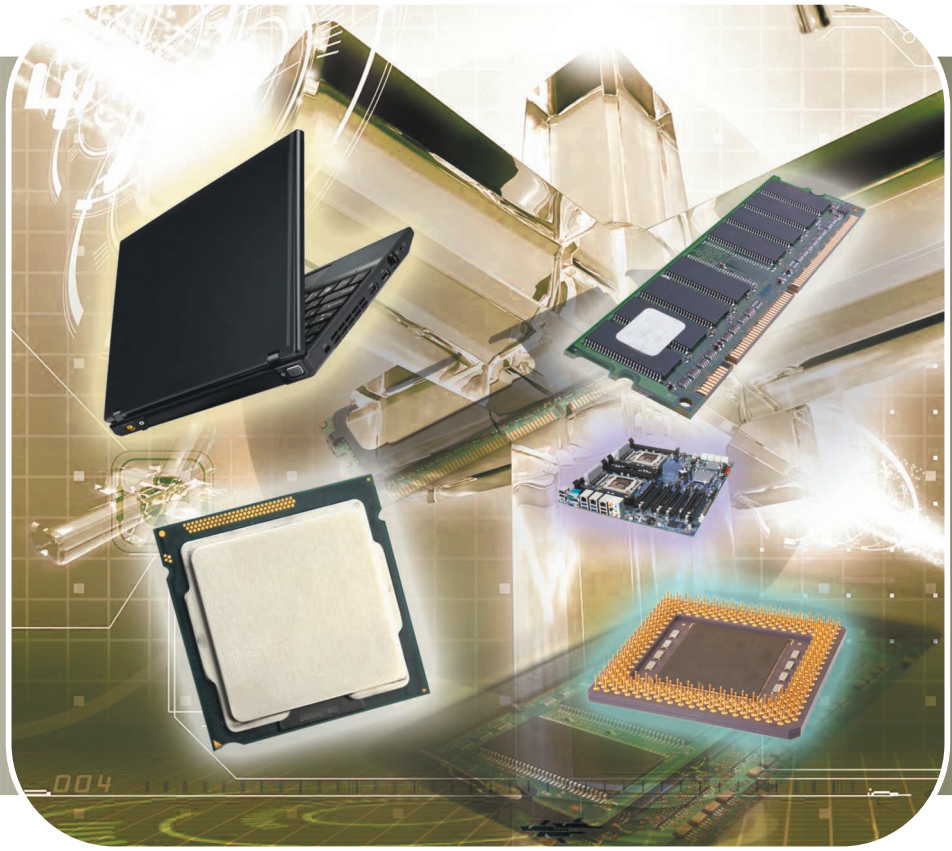
Technical Publications

Computer Architecture & Organization

A Conceptual Approach



A. P. Godse, D. A. Godse

TECHNICAL PUBLICATIONS
An Up-Thrust for Knowledge

technical@vtubooks.com
www.vtubooks.com

Computer Architecture & Organization

About the Author

A. P. Godse

- Completed M.S in Software Systems with distinction from Birla Institute of Technology.
- Completed B.E. in Industrial Electronics with distinction from University of Pune in 1990.
- Worked as a Professor at Vishwakarma Institute of Technology, Pune.
- Worked as a Technical Director at Noble Institute of Technology, Pune.
- Worked as selection Committee member for M. S. admission for West Virginia University, Washington D.C.
- Developed Microprocessor Based Instruments in co-ordination with Anna Hazare for Environmental Studies Laboratory, at Ralegan Siddhi.
- Developed Microprocessor Lab in-house for Vishwakarma Institute of Technology.
- Worked as Subject Expert for a State Level Technical Paper Presentation Competition, Pune.
- Awarded on 26th Jan 2001 by Pune Municipal Corporation for contributing in education field and technical writing.
- Awarded as a "Parvati Bhushan Puraskar" for contributing in the education field.
- Since 1996, writing books on various engineering subjects. Over the years, many of books are recommended as the reference books and text books in various national and international engineering universities.

D. A. Godse

- Completed M.E and pursuing Ph.D. in Computer Engineering from Bharati Vidyapeeth's University Pune.
- Completed B.E. in Industrial Electronics from University of Pune in 1992.
- Working as a Professor and Head of Information Technology Department in B.V.C.O.E.W, Pune.
- Subject Expert for syllabus setting of Computer Engineering and Information Technology branches at the faculty of Engineering of Pune University.
- Subject Expert and Group Leader for syllabus setting of Electronics, Electronics and Telecommunication and Industrial Electronics branches at the faculty of Maharashtra State, Board of Technical Education.
- Subject In-charge for Laboratory Manual Development, Technical Teacher's Training Institute, Pune.
- Subject In-charge for Question Bank Development Project, Technical Teacher's Training Institute, Pune.
- Subject In-charge for the preparation of Teacher's Guide, Board of Technical Examination, Maharashtra state.
- Subject Expert for a State Level Technical Paper Presentation Competition organized by Bharati Vidyapeeth's Jawaharlal Nehru Institute of Technology, Pune.
- Local Inquiry Committee (LIC) member of Engineering faculty of Pune University.
- Awarded on 15th August 2006 by Pune Municipal Corporation for contributing in education field and technical writing.
- Awarded on the occasion of International Women's Day at Yashawantrao Chavan Pratishthan Sabhagrih, Mumbai by Bharatiya Shikshan Sanstha.

COMPUTER ARCHITECTURE & ORGANIZATION

Atul P. Godse

M. S. Software Systems (BITS Pilani)
B.E. Industrial Electronics
Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Mrs. Deepali A. Godse

B.E. Industrial Electronics, M. E. (Computer)
Head of Information Technology Dept.,
Bharati Vidyapeeth's College of Engineering for Women,
Pune

TECHNICAL PUBLICATIONS
An Up-Thrust for Knowledge

ISBN 978-93-5099-234-0

Pune Nashik Bangalore Chennai Hyderabad
Ahmedabad Bhopal Lucknow Jaipur Delhi

COMPUTER ARCHITECTURE & ORGANIZATION

First Edition : January 2014
Reprint : January 2015
Reprint : March 2019

© Copyright with Author

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :

TECHNICAL PUBLICATIONS™ Amit Residency, Office No.1, 412, Shaniwar Peth, Pune - 411030, M.S. INDIA.
Ph.: +91-020-24495496/97, Telefax : +91-020-24495497
Email : technical@vtubooks.com Website : www.technicalpublications.org

ISBN 978-93-5099-234-0

ISBN 9789350992340 (Printed Book)

9789350992340 [1]

PREFACE

The importance of **Computer Architecture and Organization** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Computer Architecture and Organization**.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors
A. P. Godse
D. A. Godse

Dedicated to God.

Table of Contents

Chapter - 1 Introduction (1 - 1) to (1 - 136)

1.1 Computing and Computers ..... 1 - 2

1.1.1 Computer Types ..... 1 - 2

1.1.2 Functional Units (Elements) of Computer ..... 1 - 4

1.1.2.1 Input Unit ..... 1 - 4

1.1.2.2 Memory Unit ..... 1 - 5

1.1.2.3 Arithmetic and Logic Unit ..... 1 - 5

1.1.2.4 Output Unit ..... 1 - 5

1.1.2.5 Control Unit. ..... 1 - 6

1.1.2.6 Features of Von Neumann Model ..... 1 - 6

1.1.3 Limitations of Computers ..... 1 - 6

1.2 Evolution of Computers ..... 1 - 7

1.2.1 First Generation (Von Neumann Architecture) ..... 1 - 8

1.2.2 Detail Structure of IAS/Von Neumann Machine ..... 1 - 10

1.2.3 Second Generation ..... 1 - 15

1.2.4 Third Generation. ..... 1 - 16

1.2.5 Later Generations ..... 1 - 20

1.3 VLSI Era ..... 1 - 20

1.3.1 Integrated Circuits ..... 1 - 20

1.3.2 IC Families ..... 1 - 21

1.3.3 Processor Architecture ..... 1 - 22

1.3.4 Performance ..... 1 - 24

1.3.4.1 Processor Clock ..... 1 - 25

1.3.4.2 CPU Time ..... 1 - 26

1.3.4.3 Performance Metrics . . . . . 1 - 26
1.3.4.4 Performance Measurement. . . . . 1 - 31
1.4 Design .....1 - 34
1.4.1 System Design . . . . . 1 - 34
1.4.1.1 System Representation . . . . . 1 - 34
1.4.1.2 Design Process . . . . . 1 - 35
1.5 Register Level.....1 - 37
1.5.1 Register Level Components . . . . . 1 - 37
1.5.2 Programmable Logic Devices . . . . . 1 - 54
1.5.3 Field Programmable Gate Arrays. . . . . 1 - 61
1.5.4 Register Level Design . . . . . 1 - 63
1.5.4.1 Data and Control . . . . . 1 - 64
1.5.4.2 A Description Language . . . . . 1 - 65
1.5.4.3 Design Techniques . . . . . 1 - 66
1.6 Processor Level .....1 - 67
1.6.1 Processor Level Components. . . . . 1 - 67
1.6.1.1 Central Processing Unit . . . . . 1 - 67
1.6.1.2 Memories . . . . . 1 - 69
1.6.1.3 IO Devices . . . . . 1 - 70
1.6.1.4 Interconnection Networks . . . . . 1 - 71
1.6.2 Processor-Level Design. . . . . 1 - 72
1.6.2.1 Prototype Structures . . . . . 1 - 73
1.6.2.2 Queueing Models. . . . . 1 - 75
1.7 CPU Organization.....1 - 78
1.7.1 CPU Register Organization . . . . . 1 - 83
1.8 Data Representation.....1 - 87
1.8.1 Decimal Number System . . . . . 1 - 91
1.8.2 Binary Number System. . . . . 1 - 92
1.8.3 Octal Number System. . . . . 1 - 92
1.8.4 Hexadecimal Number System . . . . . 1 - 93
1.8.5 Format of a Binary Number . . . . . 1 - 94

1.9 Fixed Point Numbers ..... 1 - 94
1.9.1 Signed Binary Numbers ..... 1 - 95
1.9.2 1's Complement Representation ..... 1 - 96
1.9.3 2's Complement Representation ..... 1 - 96
1.9.4 Sign Extension ..... 1 - 98
1.10 Floating Point Numbers ..... 1 - 99
1.10.1 IEEE Standard for Floating-Point Numbers ..... 1 - 100
1.10.2 Special Values ..... 1 - 106
1.10.3 Exceptions ..... 1 - 106
1.11 Instruction Formats ..... 1 - 107
1.12 Instruction Types ..... 1 - 113
1.12.1 Data Transfer Instructions ..... 1 - 113
1.12.2 Arithmetic Instructions ..... 1 - 113
1.12.3 Logical Instructions ..... 1 - 115
1.12.4 Shift and Rotate Instructions ..... 1 - 115
1.12.5 Program Sequencing and Control Instructions ..... 1 - 118
1.12.6 RISC Instructions ..... 1 - 120
1.13 Addressing Modes ..... 1 - 121
1.13.1 Implementation of Variables and Constants ..... 1 - 121
1.13.2 Indirection and Pointers ..... 1 - 121
1.13.3 Indexing and Arrays ..... 1 - 122
1.13.4 Relative Addressing ..... 1 - 123
1.13.5 Additional Modes ..... 1 - 123
Two Marks Questions with Answers ..... 1 - 127

Chapter - 2 Data Path Design (2 - 1) to (2 - 106)

2.1 Introduction ..... 2 - 2
2.2 Fixed Point Arithmetic ..... 2 - 2
2.2.1 Addition ..... 2 - 2
2.2.2 Subtraction ..... 2 - 3

2.3 Adder and Subtractor Circuits ..... 2 - 9
2.3.1Half Adders ..... 2 - 9
2.3.2 Full Adders ..... 2 - 10
2.3.3 Parallel Adder ..... 2 - 12
2.3.4 Parallel Subtractor ..... 2 - 13
2.3.5 Addition / Subtraction Logic Unit ..... 2 - 14
2.3.6 Overflow in Integer Arithmetic ..... 2 - 14
2.3.7 Addition and Subtraction of Signed-magnitude Data ..... 2 - 16
2.4 Look Ahead Carry Adders ..... 2 - 18
2.5 Multiplication ..... 2 - 24
2.6 Robertson Algorithm ..... 2 - 26
2.6.1 Robertson Algorithm for 2's Multiplier ..... 2 - 26
2.6.2 Robertson Algorithm for 2's Complement Fraction ..... 2 - 30
2.7 Booth's Algorithm ..... 2 - 35
2.8 Fast Multiplication ..... 2 - 42
2.8.1 Bit-Pair Recoding of Multipliers ..... 2 - 42
2.8.2 Array Multiplier ..... 2 - 45
2.9 Division ..... 2 - 51
2.9.1 Restoring Division ..... 2 - 52
2.9.2 Non-restoring Division ..... 2 - 56
2.9.3 Comparison between Restoring and Non-restoring Division Algorithm ..... 2 - 61
2.10 Floating Point Arithmetic ..... 2 - 61
2.10.1 Addition and Subtraction ..... 2 - 61
2.10.2 Problems in Floating Point Arithmetic ..... 2 - 63
2.10.3 Flowchart and Algorithm for Floating Point Addition and Subtraction ..... 2 - 64
2.10.4 Implementing Floating-Point Operations ..... 2 - 66
2.10.5 Multiplication and Division ..... 2 - 67
2.10.6 Guard Bits and Truncation ..... 2 - 70
2.11 Arithmetic Logic Unit ..... 2 - 72

2.11.1 Design of Arithmetic Unit . . . . . 2 - 72
2.11.2 Design of Logic Circuit . . . . . 2 - 76
2.11.3 Combining Arithmetic and Logic Units . . . . . 2 - 78
2.11.4 Status Register . . . . . 2 - 81
2.11.5 Sequential ALUs . . . . . 2 - 84
2.11.6 ALU Expansion . . . . . 2 - 86
2.12 Coprocessors ..... 2 - 87
2.13 Pipeline Processing ..... 2 - 91
2.14 Pipeline Design ..... 2 - 95
Two Marks Questions with Answers ..... 2 - 100

Chapter - 3 Control Design (3 - 1) to (3 - 120)

3.1 Introduction ..... 3 - 2
3.2 Some Fundamental Concepts ..... 3 - 2
3.2.1 Register Transfer . . . . . 3 - 4
3.2.2 Performing an Arithmetic or Logic Operation . . . . . 3 - 6
3.2.3 Fetching a Word from Memory . . . . . 3 - 6
3.2.4 Storing a Word in Memory . . . . . 3 - 9
3.2.5 Execution of a Complete Instruction . . . . . 3 - 10
3.2.6 Branch Instruction . . . . . 3 - 11
3.2.7 Multiple Bus Organization . . . . . 3 - 12
3.3 Hardwired Control ..... 3 - 15
3.3.1 Design Methods of Hardwired Control Unit . . . . . 3 - 17
3.3.1.1 State-table Method . . . . . 3 - 20
3.3.1.2 Delay Element Method . . . . . 3 - 21
3.3.1.3 Sequence Counter Method . . . . . 3 - 23
3.3.1.4 PLA Method . . . . . 3 - 28
3.3.2 A Complete Processor . . . . . 3 - 30
3.3.3 CPU Control Unit . . . . . 3 - 31
3.3.4 Design of Control Unit of GCD Processor using State Table . . . . . 3 - 35

3.3.5 Design of Control Unit of GCD Processor using One Hot Method . . . . . 3 - 41

3.4 Microprogrammed Control . . . . . 3 - 44

3.4.1 Microinstruction . . . . . 3 - 49

3.4.1.1 Grouping of Control Signals . . . . . 3 - 50

3.4.1.2 Techniques of Grouping of Control Signals . . . . . 3 - 51

3.4.2 Microinstruction Sequencing . . . . . 3 - 52

3.4.3 Techniques for Modification or Generation of Branch Addresses . . . . . 3 - 55

3.4.4 Microinstructions with Next Address Field . . . . . 3 - 66

3.4.5 Microinstruction Execution . . . . . 3 - 66

3.5 Comparison between Hardwired and Microprogrammed Control . . . . . 3 - 68

3.6 Bit Slicing and Bit Sliced Microprogram Sequencer . . . . . 3 - 69

3.6.1 Features of Bit Slicing . . . . . 3 - 70

3.6.2 Processor Slice-2901 . . . . . 3 - 71

3.6.3 Microprograms for 2901 . . . . . 3 - 74

3.6.4 16-bit Bit Sliced Processor . . . . . 3 - 75

3.6.5 16-Bit Multiplication . . . . . 3 - 76

3.7 Applications of Microprogramming . . . . . 3 - 78

3.8 Pipeline Control . . . . . 3 - 79

3.8.1 Principles of Pipelining . . . . . 3 - 79

3.8.2 Instruction Pipelines . . . . . 3 - 81

3.8.3 Implementation of Two-Stage Instruction Pipelining . . . . . 3 - 83

3.8.4 Implementation of Four-Stage Instruction Pipelining . . . . . 3 - 84

3.8.5 Pipeline Performance . . . . . 3 - 85

3.8.6 Hazards in Instruction Pipelining . . . . . 3 - 90

3.8.6.1 Structural Hazards . . . . . 3 - 91

3.8.6.2 Data Hazards . . . . . 3 - 92

3.8.6.3 Instruction or Control Hazards . . . . . 3 - 95

3.8.7 Influence on Instruction Sets . . . . . 3 - 98

3.8.7.1 Addressing Modes . . . . . 3 - 98

3.8.7.2 Condition Codes . . . . . 3 - 100
3.8.8 Branch Prediction . . . . . 3 - 101
3.9 Superscaling Processing . . . . . 3 - 103
3.9.1 Instruction Level Parallelism and Machine Parallelism . . . . . 3 - 105
3.9.2 Instruction-Issue Policy . . . . . 3 - 106
3.9.3 Register Renaming . . . . . 3 - 108
3.9.4 Branch Prediction . . . . . 3 - 108
3.10 Nano Programming . . . . . 3 - 108
Two Marks Questions with Answers . . . . . 3 - 110

Chapter - 4 Memory Organization (4 - 1) to (4 - 120)

4.1 Basic Concepts . . . . . 4 - 2
4.2 Memory Hierarchy/Multilevel Memory Systems . . . . . 4 - 5
4.3 Random Access Memories . . . . . 4 - 8
4.3.1 ROM (Read Only Memory) . . . . . 4 - 8
4.3.1.1 ROM . . . . . 4 - 8
4.3.1.2 PROM (Programmable Read Only Memory) . . . . . 4 - 9
4.3.1.3 EPROM (Erasable Programmable Read Only Memory) . . . . . 4 - 10
4.3.1.4 EEPROM (Electrically Erasable Programmable Read Only Memory) . . . . . 4 - 11
4.3.2 RAM (Random Access Memory) . . . . . 4 - 11
4.3.2.1 Static RAM . . . . . 4 - 11
4.3.2.2 Internal Organization . . . . . 4 - 13
4.3.2.3 Dynamic RAM (DRAM) . . . . . 4 - 16
4.3.2.4 Internal Structure of DRAM . . . . . 4 - 17
4.3.3 Comparison between SRAM and DRAM . . . . . 4 - 17
4.4 RAM Interfaces . . . . . 4 - 18
4.4.1 Expanding Word Size and Memory Capacity . . . . . 4 - 20
4.5 Advanced DRAMs . . . . . 4 - 25
4.5.1 Enhanced DRAM (EDRAM) . . . . . 4 - 25
4.5.2 Cache DRAM (CDRAM) . . . . . 4 - 26

4.5.3 Synchronous DRAM (SDRAM) . . . . . 4 - 26
4.5.3.1 Timing Diagram . . . . . 4 - 28
4.5.3.2 Performance Measures . . . . . 4 - 28
4.5.3.3 Double-Data-Rate SDRAM . . . . . 4 - 29
4.5.4 Rambus DRAM . . . . . 4 - 30
4.5.5 Ramlink DRAM . . . . . 4 - 30
4.5.6 EDORAM . . . . . 4 - 31
4.6 Serial Access Memories . . . . . 4 - 32
4.6.1 Magnetic Disk Memory . . . . . 4 - 33
4.6.1.1 Magnetic Surface Recording . . . . . 4 - 33
4.6.1.2 Data Organization and Formatting . . . . . 4 - 34
4.6.1.3 Characteristics . . . . . 4 - 36
4.6.1.4 Typical Disks . . . . . 4 - 36
4.6.1.5 Access Time . . . . . 4 - 36
4.6.1.6 Data Buffer/Cache . . . . . 4 - 37
4.6.1.7 Disk Controller . . . . . 4 - 37
4.6.1.8 Loading of Operating System from Disk . . . . . 4 - 39
4.6.2 Floppy Disk Memory . . . . . 4 - 40
4.6.2.1 Specifications of Floppy Disks . . . . . 4 - 40
4.6.2.2 Disk Format . . . . . 4 - 40
4.6.2.3 Storage Density . . . . . 4 - 41
4.6.3 Magnetic Tape . . . . . 4 - 43
4.6.4 RAID . . . . . 4 - 47
4.6.5 Optical Memories . . . . . 4 - 50
4.6.5.1 CD-ROM . . . . . 4 - 50
4.6.5.2 WORM . . . . . 4 - 54
4.6.5.3 Erasable Optical Disk . . . . . 4 - 55
4.6.5.4 DVD Technology . . . . . 4 - 56
4.7 Cache Memory . . . . . 4 - 60
4.7.1 Cache Memory System . . . . . 4 - 61
4.7.1.1 Program Locality . . . . . 4 - 64

4.7.1.2 Locality of Reference . . . . . 4 - 64
4.7.1.3 Block Fetch . . . . . 4 - 65
4.7.2 Elements of Cache Design . . . . . 4 - 65
4.7.3 Mapping . . . . . 4 - 66
4.7.3.1 Direct-Mapping . . . . . 4 - 67
4.7.3.2 Associative-Mapping (Fully-Associative Mapping) . . . . . 4 - 68
4.7.3.3 Set-Associative Mapping . . . . . 4 - 69
4.7.4 Comparison between Mapping Techniques . . . . . 4 - 76
4.7.5 Cache Write / Updating . . . . . 4 - 76
4.7.5.1 Write through System . . . . . 4 - 76
4.7.5.2 Buffered Write through System . . . . . 4 - 77
4.7.5.3 Write-Back System . . . . . 4 - 77
4.7.6 Cache Coherency . . . . . 4 - 77
4.7.7 Replacement Algorithms . . . . . 4 - 78
4.8 Performance Issues . . . . . 4 - 80
4.9 Virtual Memory . . . . . 4 - 85
4.9.1 Address Mapping using Pages . . . . . 4 - 86
4.9.2 Page Replacement . . . . . 4 - 87
4.9.3 Page Replacement Algorithms . . . . . 4 - 88
4.9.4 Memory Management Hardware . . . . . 4 - 93
4.9.4.1 Segmented Page Mapping . . . . . 4 - 94
4.9.4.2 Memory Protection . . . . . 4 - 96
4.10 Memory Allocation . . . . . 4 - 98
4.11 Associative Memory . . . . . 4 - 101
4.11.1 Hardware Organization . . . . . 4 - 101
4.11.2 Read Operation . . . . . 4 - 104
4.11.3 Write Operation . . . . . 4 - 104
4.12 Memory Interleaving . . . . . 4 - 105
Two Marks Questions with Answers . . . . . 4 - 106

Chapter - 5 System Organization (5 - 1) to (5 - 138)

5.1 Introduction ..... 5 - 2

5.2 Communication Methods ..... 5 - 2

5.2.1 Buses ..... 5 - 2

5.2.1.1 Single Bus Structure ..... 5 - 3

5.2.1.2 Multiple Bus Structures ..... 5 - 4

5.2.1.3 Bus Design Parameters ..... 5 - 6

5.2.2 Bus Control ..... 5 - 8

5.2.2.1 Synchronous Input/Output Transfer ..... 5 - 8

5.2.2.2 Asynchronous Input/Output Transfer ..... 5 - 11

5.2.2.3 Strobe Control ..... 5 - 13

5.2.2.4 Handshaking ..... 5 - 13

5.2.3 Bus Interfacing ..... 5 - 15

5.2.4 Bus Arbitration ..... 5 - 18

5.2.4.1 Centralized Arbitration ..... 5 - 19

5.2.4.2 Distributed Arbitration ..... 5 - 21

5.2.5 PCI Bus ..... 5 - 25

5.2.5.1 Features ..... 5 - 26

5.2.5.2 PCI Configurations ..... 5 - 27

5.2.5.3 PCI Bus Signals ..... 5 - 28

5.2.5.4 PCI Bus Commands ..... 5 - 31

5.2.5.5 Data Transfer ..... 5 - 31

5.2.5.6 PCI Arbitration ..... 5 - 33

5.3 I/O and System Control ..... 5 - 35

5.3.1 I/O Modules ..... 5 - 35

5.3.1.1 Major Requirements of an I/O Module ..... 5 - 36

5.3.2 Programmed I/O ..... 5 - 37

5.3.2.1 I/O Addressing ..... 5 - 39

5.3.2.2 I/O Instructions ..... 5 - 41

5.3.2.3 Interfacing ..... 5 - 42

5.3.3 Interrupt Driven I/O ..... 5 - 44

5.3.3.1 Interrupt Hardware . . . . . 5 - 45
5.3.3.2 Enabling and Disabling Interrupts . . . . . 5 - 49
5.3.3.3 Handling Multiple Devices . . . . . 5 - 50
5.3.3.4 Vectored Interrupts . . . . . 5 - 51
5.3.3.5 Nested Interrupts . . . . . 5 - 52
5.3.3.6 Interrupt Priority . . . . . 5 - 52
5.3.3.7 Pipeline Interrupts . . . . . 5 - 53
5.3.3.8 Exceptions . . . . . 5 - 54
5.3.3.9 PCI Interrupts . . . . . 5 - 55
5.4 Comparison between Programmed I/O and Interrupt Driven I/O .....5 - 56
5.5 Direct Memory Access (DMA).....5 - 57
5.5.1 Hardware Controlled Data Transfer . . . . . 5 - 58
5.5.2 DMA Idle Cycle . . . . . 5 - 58
5.5.3 DMA Active Cycle . . . . . 5 - 59
5.5.4 DMA Channels. . . . . 5 - 59
5.5.5 Data Transfer Modes . . . . . 5 - 62
5.6 Interface Circuits.....5 - 67
5.6.1 Parallel Port. . . . . 5 - 67
5.6.1.1 Input Port . . . . . 5 - 68
5.6.1.2 Output Port . . . . . 5 - 70
5.6.1.3 Combined Input/Output Port . . . . . 5 - 72
5.6.1.4 Programmable Parallel Port . . . . . 5 - 73
5.6.2 Serial Port . . . . . 5 - 74
5.6.3 Comparison between Serial and Parallel Interface . . . . . 5 - 75
5.7 I/O Channel.....5 - 76
5.7.1 Characteristics of I/O Channel . . . . . 5 - 76
5.7.2 Types of I/O Channels. . . . . 5 - 76
5.7.2.1 Selector Channel . . . . . 5 - 76
5.7.2.2 Multiplexer Channel . . . . . 5 - 77
5.8 I/O Processor .....5 - 77

5.8.1 Features and Functions of IOP . . . . . 5 - 78
5.8.2 Block Diagram of IOP . . . . . 5 - 78
5.8.3 CPU and IOP Communication . . . . . 5 - 79
5.9 Operating Systems . . . . . 5 - 80
5.9.1 What is an Operating System ? . . . . . 5 - 80
5.9.2 Necessity of Operating System . . . . . 5 - 81
5.9.3 Functions of Operating System . . . . . 5 - 81
5.9.4 Types of Operating Systems. . . . . 5 - 81
5.9.4.1 Batch Operating System . . . . . 5 - 81
5.9.4.2 Multiprogramming Operating System . . . . . 5 - 82
5.9.4.3 Time Sharing Operating System . . . . . 5 - 83
5.9.4.4 Real-Time Operating Systems. . . . . 5 - 83
5.9.5 Protection . . . . . 5 - 83
5.9.5.1 I/O Protection . . . . . 5 - 84
5.9.5.2 Memory Protection . . . . . 5 - 84
5.9.5.3 CPU Protection . . . . . 5 - 85
5.9.6 Distributed Operating System . . . . . 5 - 85
5.9.7 Operating System Services. . . . . 5 - 86
5.9.8 System Calls. . . . . 5 - 87
5.9.8.1 Process and Job Control . . . . . 5 - 87
5.9.8.2 File Manipulation . . . . . 5 - 88
5.9.8.3 Device Manipulation . . . . . 5 - 88
5.9.8.4 Information Maintenance . . . . . 5 - 88
5.9.9 System Call Implementation . . . . . 5 - 89
5.9.10 How System Calls Used . . . . . 5 - 89
5.9.11 System Programs . . . . . 5 - 90
5.9.12 Protocol Layers . . . . . 5 - 91
5.9.13 Features of Unix . . . . . 5 - 91
5.9.14 Structure of Unix. . . . . 5 - 92
5.9.14.1 The Kernel . . . . . 5 - 92
5.9.14.2 The ps Command . . . . . 5 - 93

5.9.14.3 The Kernel and System Calls . . . . . 5 - 93
5.9.14.4 Running A Command : The Shell . . . . . 5 - 94
5.9.14.5 A Program Hierarchy . . . . . 5 - 94
5.9.15 Files and Directories . . . . . 5 - 95
5.9.15.1 File Creation . . . . . 5 - 96
5.9.15.2 File Systems . . . . . 5 - 97
5.9.16 Peripheral Devices : Special Files . . . . . 5 - 97
5.10 Multiprocessors . . . . . 5 - 99
5.10.1 Loosely Coupled Multiprocessors (LCS) . . . . . 5 - 100
5.10.2 Tightly Coupled Multiprocessors . . . . . 5 - 101
5.10.2.1 TCS without Private Cache . . . . . 5 - 102
5.10.2.2 TCS with Private Cache . . . . . 5 - 103
5.10.3 Processor Characteristics for Multiprocessing . . . . . 5 - 104
5.10.4 Interconnection Networks . . . . . 5 - 107
5.10.4.1 Time Shared Bus or Common Bus . . . . . 5 - 107
5.10.4.2 Cross Bar Switch . . . . . 5 - 108
5.10.4.3 Multiport Memory . . . . . 5 - 110
5.10.4.4 Multistage Interconnection Network (MIN) . . . . . 5 - 112
5.10.5 Contention Problems in Multiprocessor Systems . . . . . 5 - 116
5.10.5.1 Memory Contention . . . . . 5 - 116
5.10.5.2 Communication Contention . . . . . 5 - 116
5.10.5.3 Hot Spot Contention . . . . . 5 - 116
5.10.5.4 Techniques for Reducing Contentions . . . . . 5 - 117
5.10.6 Mutual Exclusion . . . . . 5 - 118
5.10.7 Dead Lock . . . . . 5 - 119
5.11 Fault Tolerance . . . . . 5 - 119
5.11.1 Redundancy . . . . . 5 - 120
5.11.2 Fault Tolerant Systems . . . . . 5 - 120
5.11.2.1 Static Redundancy . . . . . 5 - 120
5.11.2.2 Dynamic Redundancy . . . . . 5 - 121
5.11.3 Redundant Disk Arrays . . . . . 5 - 123

5.11.4 Fault Tolerance Measure . . . . . 5 - 123
5.11.4.1 Availability . . . . . 5 - 123
5.11.4.2 Reliability . . . . . 5 - 123
5.11.4.3 System Reliability . . . . . 5 - 125
5.11.4.4 Mean Time To Failure (MTTF) . . . . . 5 - 126
5.11.4.5 Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR) . . . . 5 - 127
5.11.5 Fault Tolerant Computers . . . . . 5 - 127
5.11.5.1 The Tandem Nonstop . . . . . 5 - 127
5.11.5.2 The Tandem VLX . . . . . 5 - 128
5.11.5.3 The Tandem Himalaya . . . . . 5 - 128
5.11.5.4 Architectural Details of Tandem's NonStop Computers . . . . . 5 - 128
5.11.5.5 Architectural Details of the Tandem Himalaya Computers . . . . . 5 - 130
Two Marks Questions with Answers ..... 5 - 131

Chapter - 6 RISC, CISC, Superscalar and Vector Processors (6 - 1) to (6 - 58)

6.1 RISC Processor ..... 6 - 2
6.2 RISC Versus CISC ..... 6 - 2
6.3 RISC Properties ..... 6 - 4
6.4 RISC Addressing Modes ..... 6 - 7
6.5 RISC Evaluations ..... 6 - 8
6.5.1 RISC and VLSI Realization . . . . . 6 - 8
6.5.2 Computing Speed . . . . . 6 - 9
6.5.3 Design Cost and Reliability Considerations . . . . . 6 - 9
6.5.4 HLL Support . . . . . 6 - 10
6.5.5 Shortcomings of RISC Architecture . . . . . 6 - 10
6.6 On-chip Register File Versus Cache Evaluation ..... 6 - 11
6.7 Overview of RISC Development and Current Systems ..... 6 - 13
6.7.1 SPARC . . . . . 6 - 13
6.7.2 The Intel i860 Processor Architecture . . . . . 6 - 17

6.7.3 RISC Processor-Motorola 88000 . . . . . 6 - 20

6.8 Superscalar Processor . . . . . 6 - 27

6.8.1 Instruction Level Parallelism and Machine Parallelism . . . . . 6 - 29

6.8.2 Instruction-Issue Policy . . . . . 6 - 30

6.8.3 Register Renaming . . . . . 6 - 32

6.8.4 Branch Prediction . . . . . 6 - 32

6.8.5 Example : PowerPC Processor . . . . . 6 - 32

6.8.6 Power Interrupts . . . . . 6 - 37

6.8.7 Machine Status Register (MSR) . . . . . 6 - 38

6.8.8 Data Types . . . . . 6 - 40

6.8.9 PowerPC Addressing Modes . . . . . 6 - 40

6.8.10 PowerPC Instruction Formats . . . . . 6 - 41

6.8.11 PowerPC Instruction Set . . . . . 6 - 42

6.8.12 Features of PowerPC Architecture . . . . . 6 - 47

6.8.13 Features Not Defined by the PowerPC Architecture . . . . . 6 - 48

6.9 Vector Processor . . . . . 6 - 49

6.9.1 Characteristics of Vector Processing . . . . . 6 - 49

6.9.2 Vector Processing Approach . . . . . 6 - 50

Two Marks Questions with Answers . . . . . 6 - 55

1 Introduction

Syllabus

Computing and computers, Evolution of computers, VLSI era, System design-register level, Processor level, CPU organization, Data representation, Fixed-point numbers, Floating point numbers, Instruction formats, Instruction types. Addressing modes.

Contents

1.1 Computing and Computers	**May-04, 07, 09, 10, Dec.-04, 07, 08, 09, 10,**	Marks 16
1.2 Evolution of Computers	**May-07, Dec.-11,**	Marks 12
1.3 VLSI Era	**Dec.-03, 08,**	Marks 16
1.4 Design		
1.5 Register Level	**Dec.-03, 11, May-11,**	Marks 10
1.6 Processor Level		
1.7 CPU Organization	**May-09,**	Marks 8
1.8 Data Representation	**Dec.-11, May-12,**	Marks 8
1.9 Fixed Point Numbers	**Dec.-07, May-09,**	Marks 2
1.10 Floating Point Numbers	**Dec.-06, May-07, 08**	Marks 6
1.11 Instruction Formats	**Dec.-03, 06, 07, 10, May-06, 08, 09,**	Marks 8
1.12 Instruction Types	**May-06, 12, Dec.-06, 12,**	Marks 10
1.13 Addressing Modes	**Dec.-06, 07, 08, 10, 11, 12, May-07, 08, 09, 10, 11,**	Marks 16

Two Marks Questions with Answers

1.1 Computing and Computers

May-04, 07, 09, 10, Dec.-04, 07, 08, 09, 10,

This chapter provides a broad overview of digital computers. This section of the chapter first examines the nature and limitations of the computing process and then explains the evolution of computers. The later part of this section discuss the VLSI based computer systems.

In early days, the man was using the abacus and the slide rules for computing purpose. But as the size and complexity of the calculations being carried out increases, these manual calculation1s has serious limitations :

- The speed at which manual calculations are carried out is limited.
- Humans are notoriously prone to error, so manually done long calculations are unreliable unless elaborate precautions are taken to eliminate mistakes.

On the other hand, the computer is a machine which do not have distraction and fatigue and can perform billions of operations in considerable quick time. Further more, they can provide results free from error.

1.1.1 Computer Types

A digital computer or simply computer in its simplest form is a fast electronic calculating machine that accepts digitized information from the user, processes it according to a sequence of instructions stored in the internal storage and provides the processed information to the user. The sequence of instructions stored in the internal storage is called **computer program** and internal storage is called **computer memory**.

According to size, cost computational power and application computers are classified as :

- Microcomputers
- Minicomputers
- Desktop computers
- Personal computers
- Portable notebook computers
- Workstations
- Mainframes or enterprise systems
- Servers
- Supercomputers

Microcomputers : As the name implies microcomputers are smaller computers. They contain only one Central Processing Unit. One distinguishing feature of a microcomputer is that the CPU is usually a single integrated circuit called a microprocessor.

Microcomputer is the integration of microprocessor and supporting peripherals (memory and I/O devices). The word length depends on the microprocessor used and is in the range of 8-bits to 32-bits. These type of computers are used for small industrial control, process control and where storage and speed requirements are moderate.

Minicomputers : Minicomputers are the scaled up version of the microcomputers with the moderate speed and storage capacity. These are designed to process smaller data words, typically 32-bit words. This type of computers are used for scientific calculations, research, data processing application and many other.

Desktop Computers : The desktop computers are the computers which are usually found on a home or office desk. They consist of processing unit, storage unit, visual display and audio as output units and keyboard and mouse as input units. Usually storage unit of such computer consists of hard disks, CD-ROMs and diskettes.

Personal Computers : The personal computers are the most common form of desktop computers. They found wide use in homes, schools and business offices.

Portable Notebook Computers : Portable notebook computers are the compact version of personal computers. The laptop computers are the good example of portable notebook computer.

Workstations : Workstations have higher computation power than personal computers. They have high resolution graphics terminals and improved input/output capabilities. Workstations are used in engineering applications and in interactive graphics applications.

Mainframes or Enterprise Systems : Mainframe computers are implemented using two or more central processing units (CPU). These are designed to work at very high speeds with large data word lengths, typically 64-bits or greater. The data storage capacity of these computers is very high. This type of computers are used for complex scientific calculations, large data processing applications, Military defense control and for complex graphics applications (e.g.: For creating walkthroughs with the help of animation softwares).

Servers : These computers have large storage unit and faster communication links. The large storage unit allows to store sizable database and fast communication links allow faster communication of data blocks with computers connected in the network. These computers serve major role in internet communication.

Supercomputers : These computers are basically multiprocessor computers used for the large-scale numerical calculations required in applications such as weather forecasting, robotic engineering, aircraft design and simulation.

1.1.2 Functional Units (Elements) of Computer

The idea of having computer wired for general computations with program stored in memory was introduced by John Von Neumann when he was working as a consultant at the Moore school. He and originators of ENIAC designed the first *stored program computer* named EDVAC (Electronic Discrete Variable Computer). The two key principles are used to build computers :

1. Instructions are represented as numbers.
2. Programs can be stored in memory to be read or written just like numbers.

These principles lead to the stored-program concept. According to stored-program concept memory can contain the program (source code), the corresponding compiled machine code, editor program and even the compiler that generated the machine code. The Fig. 1.1.1 shows the general structure of a Von Neumann machine. It consists of five, functional units : Input unit, output unit, control unit, Arithmetic and logic unit and memory unit.

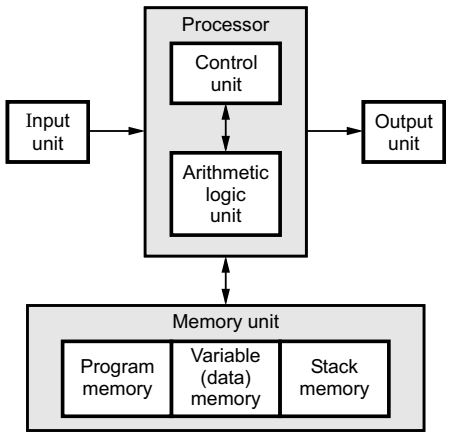


Fig. 1.1.1 A Von Neumann machine

The input unit accepts the digital information from user with the help of input devices such as keyboard, mouse, microphone etc. The information received from the input unit is either stored in the memory for later use or immediately used by the arithmetic and logic unit to perform the desired operations. The program stored in the memory decides the processing steps and the processed output is sent to the user with the help of output devices or it is stored in the memory for later reference. All the above mentioned activities are co-ordinated and controlled by the control unit. The arithmetic and logic unit in conjunction with control unit is commonly called **Central Processing Unit (CPU)**.

1.1.2.1 Input Unit

A computer accepts a digitally coded information through input unit using input devices. The most commonly used input devices are keyboard and mouse. The keyboard is used for entering text and numeric information. On the other hand mouse is used to position the screen cursor and thereby enter the information by selecting option. Apart from keyboard and mouse there are many other input devices are available, which include joysticks, trackball, spaceball, digitizers and scanners.

1.1.2.2 Memory Unit

The memory unit is used to store programs and data. Usually, two types of memory devices are used to form a memory unit : primary storage memory device and secondary storage memory device. The primary memory, commonly called main memory is a fast memory used for the storage of programs and active data (the data currently in process). The main memory is a semiconductor memory. It consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are read or written by the central processing unit in a group of fixed size called **word**. The main memory is organized such that the contents of one word, containing n bits, can be stored or retrieved in one write or read operation, respectively.

To access data from a particular word from main memory each word in the main memory has a distinct address. This allows to access any word from the main memory by specifying corresponding address. The number of bits in each word is referred to as the **word length** of the computer. Typically, the word length varies from 8 to 64 bits. The number of such words in the main memory decides the **size of memory** or **capacity of the memory**. This is one of the specification of the computer. The size of computer main memory varies from few million words to tens of million words.

An important characteristics of a memory is an **access time** (the time required to access one word). The access time for main memory should be as small as possible. Typically, it is of the order of 10 to 100 nanoseconds. This access time also depend on the type of memory. In randomly accessed memories (RAMs), fixed time is required to access any word in the memory. However, in sequential access memories this time is not fixed.

The main memory consists of only randomly accessed memories. These memories are fast but they are small in capacities and expensive. Therefore, the computer uses the secondary storage memories such as magnetic tapes, magnetic disks for the storage of large amount of data.

1.1.2.3 Arithmetic and Logic Unit

The arithmetic and logic unit (ALU) is responsible for performing arithmetic operations such as add, subtract, division and multiplication, and logical operations such as ANDing, ORing, Inverting etc. To perform these operations operands from the main memory are brought into the high speed storage elements called **registers** of the processor. Each register can store one word of data and they are used to store frequently used operands. The access times to registers are typically 5 to 10 times faster than access times to memory. After performing operation the result is either stored in the register or memory location.

1.1.2.4 Output Unit

The output unit sends the processed results to the user using output devices such as video monitor, printer, plotter, etc. The video monitors display the output on the CRT screen whereas printers and plotters give the hard-copy output. Printers are classified

according to their printing methodology : **Impact printers** and **non-impact printers**. Impact printers press formed character faces against an inked printers. Non impact printers and plotters use laser techniques, ink-jet sprays, xerographic processes, electrostatic methods, and electrothermal methods to get images onto the paper. A ink-jet printer are the examples of non-impact printers.

1.1.2.5 Control Unit

As mentioned earlier, the control unit co-ordinates and controls the activities amongst the functional units. The basic function of control unit is to fetch the instructions stored in the main memory, identify the operations and the devices involved in it, and accordingly generate control signals to execute the desired operations.

The control unit uses control signals or timing signals to determine when a given action is to take place. It controls input and output operations, data transfers between the processor, memory and input/output devices using timing signals.

The control and the arithmetic and logic units of a computer are usually many times faster than other devices connected to a computer system. This enables them to control a number of external input/output devices.

1.1.2.6 Features of Von Neumann Model

The main features of a Von Neumann model are :

1. It uses stored program concept. The program (instructions) and data are stored in a single read-write memory.
2. The contents of read-write memory are addressable by location, without regard to the type of data contained there.
3. Execution of instructions occurs in a sequential manner (unless explicitly modified) from one instruction to the next.

Because of the stored program architecture of Von-Neumann machine, the processor performance is tightly bound to the memory performance. That is, since we need to access memory at least once per cycle to read an instruction, the processor can only operate as fast as the memory. This is sometimes known as the **Von Neumann bottleneck** or **memory wall**.

1.1.3 Limitations of Computers

1. Unsolvable problems : The problems which has no known solution procedure or has no solution cannot be solved by a computer.

There are some problems which can be solved for specific input conditions however there is no general procedure to solved the problem which can accept all the possible inputs. Such problems are known as **undecidable problems.** Such undecidable problems cannot be solved by computers.

2. Intractable problems : The problem which can be solved by computer of reasonable size and cost with acceptable degree of accuracy and in a reasonable amount of time is called **tractable problems.** The problems which are not tractable are called **intractable problems.** We normally say a problem is intractable if all its known solution methods grow exponentially with the size of problem.

An intractable problem can be solved in a reasonable amount of time only when its size n is below some maximum value n_{max}. The size n depends on the speed of the computer.

3. Speed limitations : There are limitations on the speed of computer. Although computers continue to increase in speed because of advances in the hardware technology, the rate of increase has not kept pace with demand. Thus it is necessary to find new ways to improve the performance of computer at reasonable cost.

Review Questions

1. *List various elements of computer.*
2. *Explain the function of each functional unit in the computer system with suitable diagram.* **Dec.-09, May-09, Dec.-07, Marks 16**
3. *Name the functional units of a computer and how they are interrelated.* **Dec.-08, Marks 2**
4. *With a neat diagram explain Von Neumann computer architecture.* **May-07, Marks 12**
5. *What is meant by the stored program concept ? Discuss.* **May-07, Marks 2**
6. *What is meant by Central Processing Unit (CPU) ?* **Dec.-09, Marks 2**
7. *Draw and explain the block diagram of a simple computer with five functional units.* **Dec.-04, May-04, Marks 8**
8. *What is meant by the stored program concept ? Discuss.* **May-07, Marks 2**
9. *Explain in detail about functional unit of computers.* **Dec.-07, Marks 16**
10. *Name the functional units of a computer and how they are interrelated.* **Dec.-08, Marks 2**
11. *Describe the functional units of the computer system.* **May-09, Marks 8**
12. *What is meant by Central Processing Unit(CPU)?* **May-10, Marks 2**
13. *Explain the architecture of basic computer with neat diagram.* **Dec.-10, Marks 16**

1.2 Evolution of Computers

May-07, Dec.-11

In 1623-62 the french philosopher Blaise Pascal invented an early and influential mechanical calculator that could add and subtract decimal numbers. In the 19th century Charles Babbage designed the first mechanical computer that can perform multistep operations automatically, that is, without a human intervening in every step. However, these mechanical machines suffered from two serious drawbacks :

- Their computing speed was limited by the inertia of their moving parts and
- The transmission of digital information by mechanical means was quite unreliable.

The further research in this field invented the electronic computer, in which the "moving parts" are electrons and they can be transmitted and processed reliably at speeds approaching that of light (300,000 km/s).

1.2.1 First Generation (Von Neumann Architecture)

The first electronic computer, ***ENIAC*** (Electronic Numerical Integrator And Computer), was designed and constructed under the direction of Eckert and Mauchly at the Moore School of Engineering (University of Pennsylvania). It was made up of more than 18000 vacuum tubes and 1500 relays. ENIAC's primary function was to compute ballistic trajectories. It was able to perform nearly 5000 additions or subtractions per second.

The ENIAC was a decimal rather than a binary machine. All numbers in an ENIAC were represented in decimal form and arithmetic was performed in the decimal system. Its data memory consists of 20 "*accumulators*", each capable of storing a ten digit decimal number. Each digit was represented by a ring of 10 vacuum tubes and only one vacuum tube was in the ON state to represent one of the ten digits. The major drawback of the ENIAC was that it was wired in for specific computations. For modifications and replacements of programs manually setting of switches and plugging and unplugging of cables was necessary. It was a very time consuming process. Despite these shortcomings, ENIAC was used for about ten years.

The idea of having computer wired for general computations with program stored in memory was introduced by John Von Neumann when he was working as a consultant at the Moore school. He and originators of ENIAC designed the first *stored program computer* named EDVAC (Electronic Discrete Variable Computer). The stored program concept in EDVAC facilitated the users to enter and alter various programs and do variety of computations.

The EDVAC project was further developed by Von Neumann with his collaborators at the Institute for Advanced Studies (IAS) in Princeton. They came up with a new machine referred to as IAS or Von Neumann Machine. It has now become the usual frame of reference for many modern computers.

Fig. 1.2.1 shows the general structure of a Von Neumann machine. It

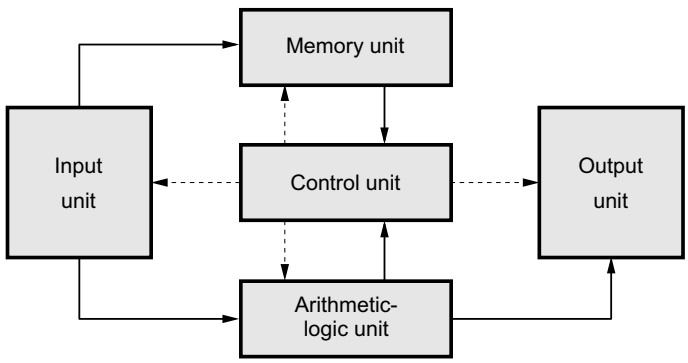


Fig. 1.2.1 A Von Neumann machine

consists of five basic units whose functions can be summarized as follows :

- The input unit transmits data and instructions from the outside world to machine. It is operated by control unit.
- The memory unit stores both, data and instructions.
- The arithmetic-logic unit performs arithmetic and logical operations.
- The control unit fetches and interprets the instructions in memory and causes them to be executed.
- The output unit transmits final results and messages to the outside world.

In the original IAS machine (Von Neumann machine), memory unit consists of 4096 storage locations ($2^{12} = 4096$) of 40 bits each, referred to as words. These memory locations are used to store data as well as instructions. Both, data and instructions are represented in the binary form with a specific format as shown in the Fig. 1.2.2.

Data Format :

Fig. 1.2.2 shows data format. The leftmost bit (bit 0) represents the sign of the number (0 for positive and 1 for negative), while the remaining 39 bits indicate the number's size in a two's complement form.

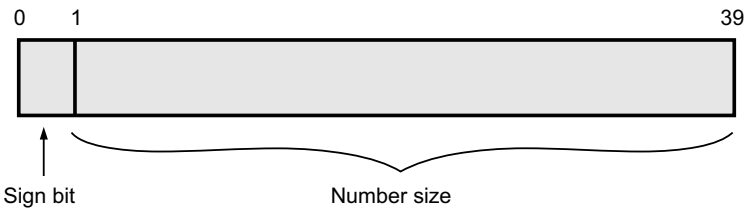


Fig. 1.2.2 Data format

The numbers are assumed to have an implicit binary point corresponding to the decimal point in ordinary decimal notation. It may be placed in any fixed position within the number word format ; hence these numbers are called *fixed-point*. If the implicit binary point is assumed to lie between bits 0 and 1, then all numbers are treated as fractions. Some examples of the IAS representation of fractions are as follows :

+ 0.5 = 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000

+ 0.1 = 0000 1100 1100 1100 1100 1100 1100 1100 1100 1100

Zero = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

– 0.1 = 1111 0011 0011 0011 0011 0011 0011 0011 0011 0100

With binary point fixed between bits 0 and 1, fractions are restricted to lie between –1 and +1. Hence, all numbers used in calculations that lie outside this range must be adjusted by some suitable scaling factor.

Instruction format : Fig. 1.6 shows the instruction format. Each instruction is of 20-bits, so that two instructions can be stored in each 40-bits memory location. As shown in Fig. 1.2.3 instruction consists of two parts : an 8-bit opcode (operation code) and 12-bit address. The opcode defines the operation to be performed (add, subtract, etc.) and address part identifies any of the 2^{12} memory locations that may be used to store an operand of the instruction.

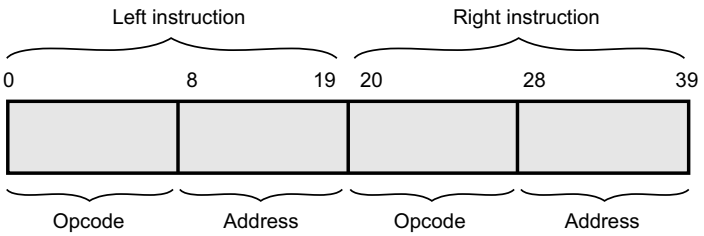
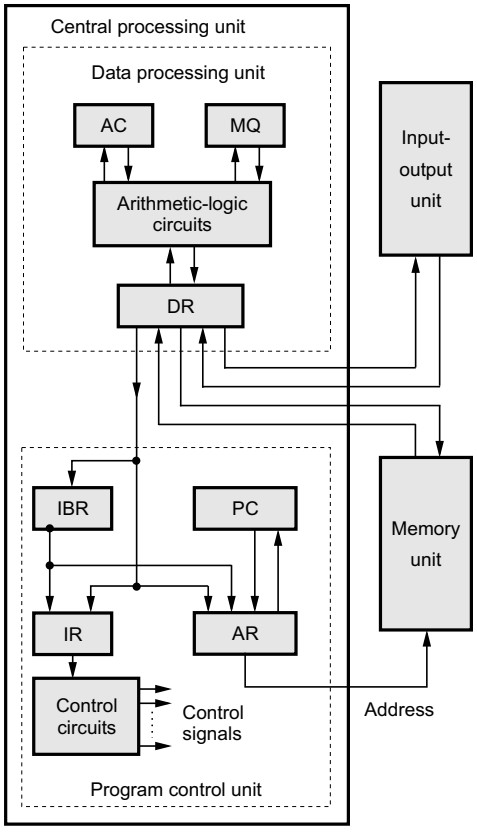


Fig. 1.2.3 Instruction format

1.2.2 Detail Structure of IAS/Von Neumann Machine

Structure



Register	Abbreviation
Accumulator	AC
Multiplier Quotient	MQ
Data Register	DR
Program Counter	PC
Instruction Buffer Register	IBR
Address Register	AR
Instruction Register	IR

Fig. 1.2.4 Structure of IAS computer

Fig. 1.2.4 shows detail structure of IAS computer. It consists of various processing and control units, along with a set of high speed registers (AC, MQ, DR, IBR, PC, IR, and AR). These registers are used to store instructions, memory addresses and data.

The complete instruction cycle involves three operations : Instruction fetching, opcode decoding and instruction execution. The control circuits in the program control unit are responsible for fetching instructions, decoding opcodes, routing information correctly through the system and providing proper control signals for central processing unit (CPU) actions. After decoding, the arithmetic logic circuits of the data processing unit perform actions specified by the instruction. An electronic clock circuit (not shown in the Fig. 1.2.1) is used to generate the basic timing signals to synchronize the operation of the different parts of the system. The functioning of different registers is as given below :

PC (Program Counter)

It is an address register. It is used to store the address of the next instruction to be executed and hence also referred to as instruction address register.

AR (Address Register)

It is a 12-bit address register. It is used to specify the address in memory of the word to be written into or read from the DR.

DR (Data Register)

It is a 40-bit register. It is used to store any 40-bit word. A word transfer can take place between the 40-bit data register DR of the CPU and any memory location. The DR may be used to store an operand during the execution of an instruction.

AC (Accumulator) and MQ (Multiplier-Quotient)

These are two 40-bit registers used for the temporary storage of operands and results.

IR (Instruction Register) and IBR (Instruction Buffer Register)

Program control unit fetches two instructions simultaneously from memory. The opcode of the first instruction is placed in the instruction register (IR) and the instruction that is not to be executed immediately (second instruction) is placed in the instruction buffer register (IBR).

Before going to see detail operations of instruction processing, we will see the instructions of the IAS computer.

Instructions

The instructions of IAS computer are divided in five groups :

- Data transfer
- Unconditional branch
- Conditional branch
- Arithmetic
- Address modify

Table 1.2.1 shows the instruction set of IAS computer.

Instruction type	Shorthand notation	Description
Data transfer	AC ← MQ	Transfer contents of register MQ to the accumulator AC
	MQ ← M(X)	Transfer contents of memory location X to MQ
	M(X)← AC	Transfer contents of accumulator to memory location X
	AC ← M(X)	Transfer M(X) to the accumulator
	AC ← – M(X)	Transfer – M(X) to the accumulator
	AC ← | M(X)|	Transfer absolute value of M(X) to the accumulator
	AC ← – |M(X)|	Transfer – |M(X)| to the accumulator
Unconditional branch	go to M(X, 0:19)	Take next instruction from left half of M(X)
	go to M(X, 20:39)	Take next instruction from right half of M(X)
Conditional branch	if AC ≥ 0 then go to M(X, 0:19)	If number in the accumulator is non negative, take next instruction from left half of M(X)
	if AC ≥ 0 then go to M(X, 20:39)	If number in the accumulator ins nonnegative, take next instruction from right half of M(X)
Arithmetic	AC ← AC + M(X)	Add M(X) to AC; put the result in AC
	AC ← AC + |M(X)|	Add |M(X)| to AC; put the result in AC
	AC ← AC – M(X)	Subtract M(X) from AC; put the result in AC
	AC ← AC – |M(X)|	Subtract |M(X)| from AC; put the result in AC
	AC.MQ ← MQ × M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	MQ.AC ← AC ÷ M(X)	Divide AC by M(X); put the quotient in MQ and the remainder AC
	AC ← AC × 2	Multiply accumulator by 2, i.e., shift left one bit position
	AC ← AC ÷ 2	Divide accumulator by 2, i.e., shift right one bit position
Address modify	M(X, 8:19) ← AC(28:39)	Replace left address field at M(X) by 12 rightmost bits of AC
	M(X, 28:39) ← AC(28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

Table 1.2.1 Instruction set of IAS computer

Instruction Cycles

Let us see how instruction is processed. The complete instruction cycle involves three operations : Instruction fetching, opcode decoding and instruction execution.

The Fig. 1.2.5 shows the basic *instruction cycle*. After each instruction cycle, central processing unit checks for any valid interrupt request. If so, central processing unit

fetches the instructions from the interrupt service routine and after completion of interrupt service routine, central processing unit starts the new instruction cycle from where it has been interrupted. The Fig. 1.2.6 shows instruction cycle with interrupt cycle.

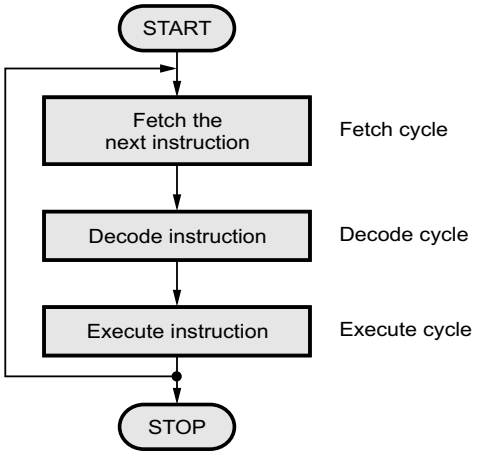


Fig. 1.2.5 Basic instruction cycle

Fig. 1.2.7 shows the principle actions carried out in each cycle.

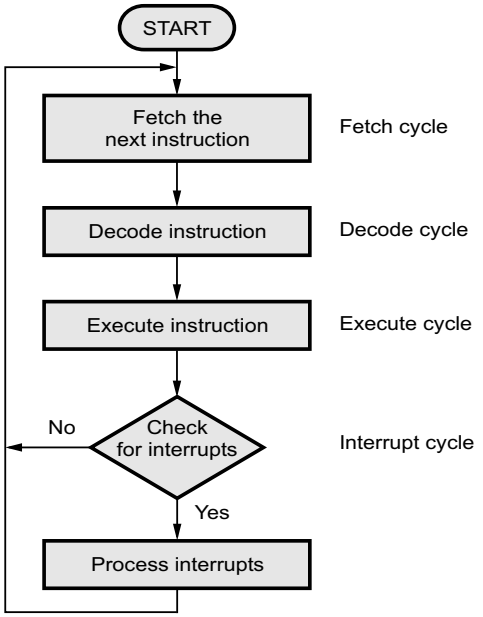


Fig. 1.2.6 Basic instruction cycle with interrupt

Fetch Cycle :

The fetch cycle is common to all instructions. We know that program control unit fetches two instructions simultaneously from memory, it is necessary to check whether the next instruction is available in the IBR or not. If not, the previously incremented contents of the program counter are transferred to the address register and a Read request is sent to Memory (M). The required data at memory location X (M(x)) is then transferred to the data register DR. The opcode of the required instruction (which is in either the left or right half of the fetched word) is sent to the instruction register and address part is sent to the address register, while the second instruction may be transferred to the instruction buffer register IBR.

If next instruction is available in the IBR, opcode part is sent to the instruction register and address part is sent to the address register.

It is important to note that program counter is incremented only when instruction is read from memory i.e. when the next instruction is not available in the IBR.

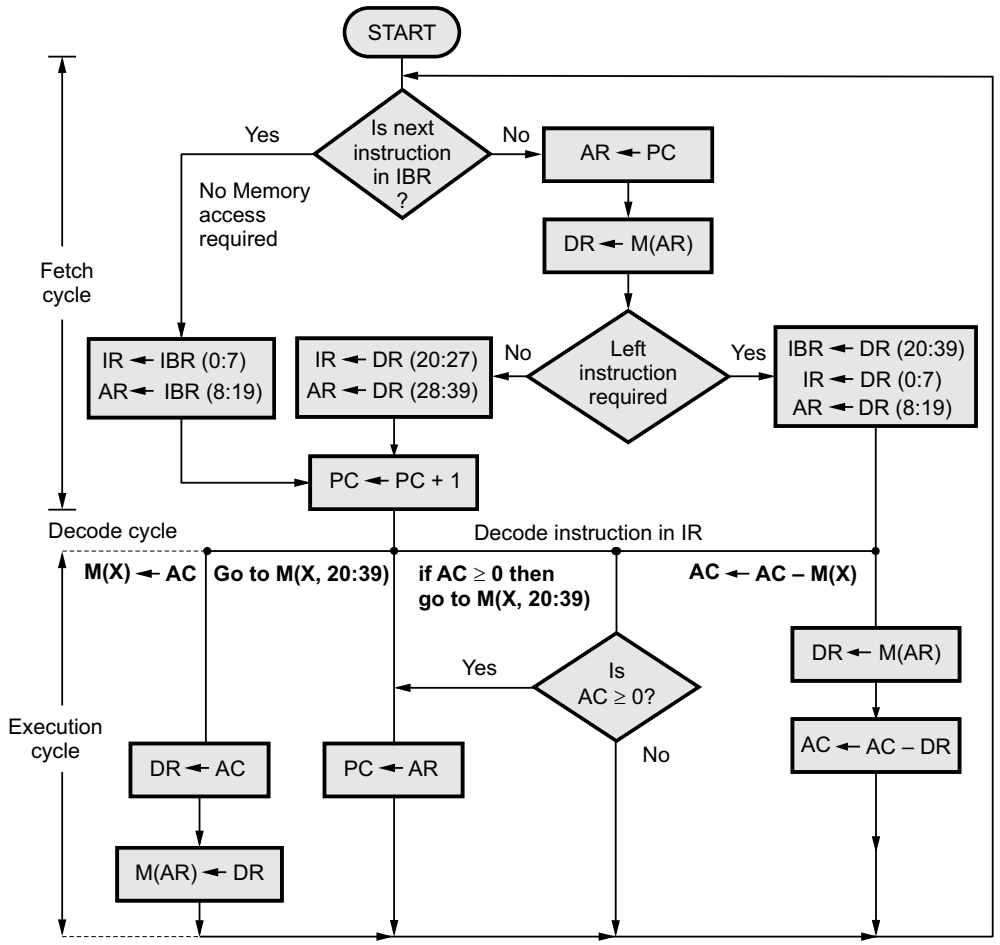


Fig. 1.2.7 Three phases of instruction cycle

Decode Cycle

In the decode cycle, the instruction in the instruction register is decoded by the control circuits in the program control unit.

Execution Cycle

In the execution cycle micro-operations depending on the instructions are carried out. Fig. 1.2.7 shows operations for four instructions: M(X) ← AC, go to M(X, 20:39), if AC ≥ 0 then go to M(X, 20:39) and AC ← AC – M(X). Note that each instruction is executed by a sequence of micro-operations. For example, instruction M(X) ← AC requires two micro-operations; first the contents of accumulator are transferred to the data register and then the contents of data register DR are transferred to the memory location specified by the address register AR.

Programming Examples

Program 1 : Subtract two numbers

```
AC ← M(50)              ; Get the contents of memory location
                        ; 50 into
                        ; the accumulator
AC ← AC – M(51)         ; Subtract the contents of memory location 51 from
                        ; the contents of accumulator and place the result in
                        ; the accumulator
M(52) ← AC              ; Store the contents of accumulator, result in the
                        ; memory location 52.
```

Program 2 : Solve equation 2a + 2b, where M(100) = a and M(101) = b

```
AC ← M(100)             ; Get the contents of memory location 100 into
                        ; the accumulator
AC ← AC × 2             ; Multiply accumulator by 2 (a × 2)
M(102) ← AC             ; Save the result of multiplication
AC ← M(101)             ; Get the contents of memory 101
                        ; into the accumulator
AC ← AC × 2             ; Multiply accumulator by 2 (b × 2)
AC ← AC + M(102)        ; Add the contents of memory
                        ; location 102 to the contents of
                        ; the accumulator and place
                        ; result (2a + 2b) in the accumulator.
```

1.2.3 Second Generation

In the second generation, IBM 7094 system, the first major change in the electronic computer came with the replacement of the vacuum tube by the transistor. From the architecture point of view, the second generation CPU differs from that of the IAS computer mainly in the addition of a set of index registers and arithmetic circuits that can handle both floating-point and fixed-point operations. They have separate I/O processors having direct access to main memory to control I/O operations.

In the second generation, magnetic core memories and magnetic drum storage devices were more widely used. In this generation, the higher level languages such as Fortran were developed, making the preparation of application programs much easier. System programs called compilers were developed to translate these high-level language programs into a corresponding assembly language program, which was then translated into executable machine language form.

The Fig. 1.2.8 shows the IBM 7094 configuration. We see the major difference is the use of data channels. A data channel is an independent I/O module with its own

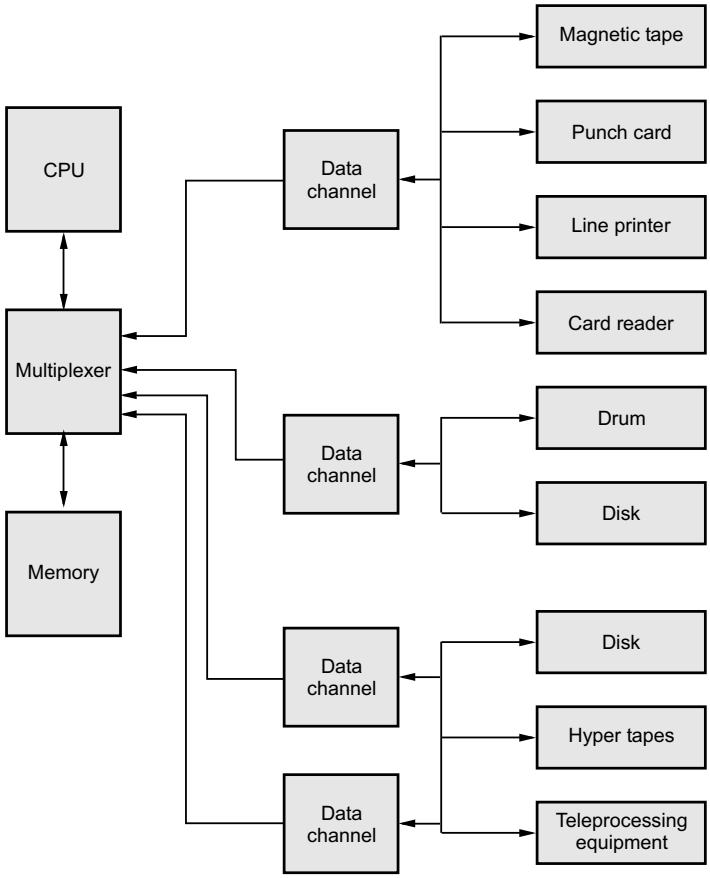


Fig. 1.2.8 IBM 7094 configuration

processor (I/O processor) and its own instruction set. A computer system with I/O processor does not execute detailed I/O instructions. Such instructions are stored in a main memory to be executed by a special purpose processor in the data channel itself. In such configuration CPU initiates an I/O transfer by sending a control signal to the data channel, instructing it to execute a sequence of instructions in memory. The data channel can perform this task independently relieving the processing burden of the CPU.

Another new feature of 7094 is the use of multiplexer. It schedules access to the memory from the CPU and data channels, allowing these devices to act independently.

1.2.4 Third Generation

In mid 1960's, Integrated Circuits (ICs) began to replace the discrete transistor circuits to introduced third generation of computers. In the third generation various techniques were introduced to improve the performance of the computer. These are as follows :

- Microprogramming
- Parallel processing
  a) Multiprocessing b) Pipelining
- Sharing resources

Microprogramming was introduced to simplify CPU design and increase their flexibility, whereas parallel processing was introduced to increase effective speed at which programs could be executed. To cope up with the demand of large memory space and to offset the speed difference of electronic circuit of CPU and memory subsystem, semiconductor memories were introduced, replacing ferrite cores.

Table 1.2.2 gives the listing of some third generation systems.

Company	Products
Burroughs Corporation	IBM 360, B5500, 6500, 7500, 8500
String Array Computer	UNIVAC 1108, CDC 6600, 7600, CDR STAR-100
Illinois Automatic Computer	ILLIAC IV
Digital Equipment Corporation	PDP-5, PDP-8, PDP-11

Table 1.2.2

In 1964, IBM had announced new series of computers (third generation), named system/360. They came up with systematic *computer architecture* and its implementation. The architecture of computer is its structure and behaviour as seen by an assembly language programmer. It includes data and instruction formats, addressing modes, instruction set and the general organization of the CPU registers, main memory and I/O system.

The implementation, on the other hand, refers to the logical and physical design techniques used to realize the architecture in any specific instance. The logical aspect of the implementation can be referred to as *computer organization*.

IBM has continued to add new computer, IBM 370 series to this family (system/360). All computers representing system/360 and system 370 families have same architecture, but they have different implementations.

Structure of IBM 360/370

Fig. 1.2.9 shows the general structure of a typical system/360-370 or IBM 360-370 computer.

As shown in Fig. 1.2.9, two separate I/O channels are used: multiplexer channels and selector channels. The multiplexer channels allow the multiplexed (interleaved) data

transmission between main memory and several I/O devices, where as selector channels allow the data transmission only with one I/O device at a time.

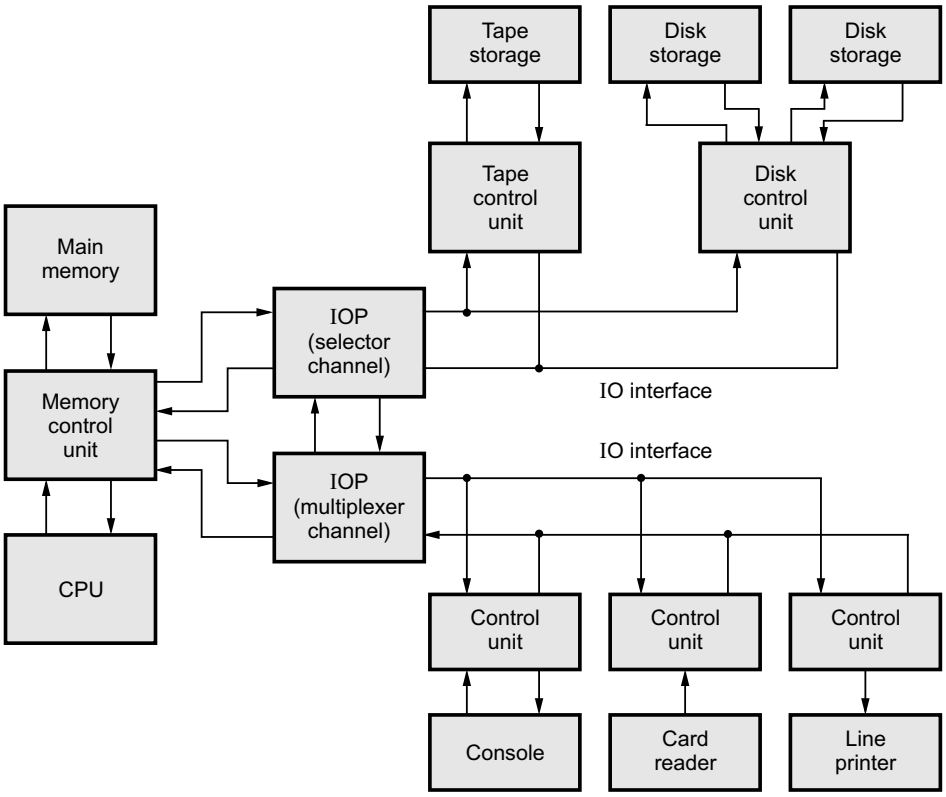


Fig. 1.2.9 Structure of an S/360-370 series computer

The selector channels are intended for use with very high speed I/O devices, such as magnetic disk while multiplexer channels are intended for controlling a number of low speed devices such as printers, card readers and card punches.

The I/O devices are interfaced to memory control unit through selector channels or multiplexer channels. The I/O devices are connected to selector or multiplexer channels with the help I/O interface bus. It consists of data and control signals.

Features of IBM 360-370

- It uses 32-bit or 4 byte word format.
- It has 16 general registers and 4 double length floating point registers.
- It provides extensive instruction set, almost 150 opcodes supported with word, byte, integer, decimal and floating point operations.

- It supports data transfer instructions for register to register, register to memory, memory to register or memory to memory data transfers.
- The main memory of the 360 is faster and can be expanded upto 1 million words in some models.

Basic CPU Implementation

Fig. 1.2.10 shows the structure of central processing unit for IBM 360-370. As shown in the Fig. 1.2.10, arithmetic-logic unit is divided logically into three subunits :

- Fixed point arithmetic unit
- Decimal arithmetic unit
- Floating point arithmetic unit

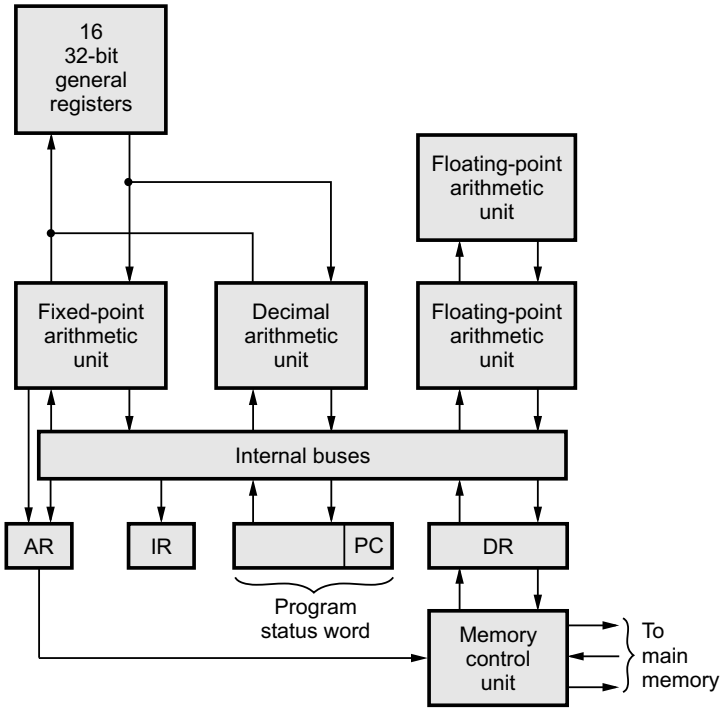


Fig. 1.2.10 CPU structure of S/360-370 series computers

It performs following operations :

- Fixed point operations, including binary integer arithmetic and effective address computation.
- Floating point arithmetic
- Variable-length operations, including decimal arithmetic and character string operations.

As mentioned earlier, 360 system consists of sixteen 32-bit general registers. These are used to store operands and results. For floating point arithmetic four 64-bit floating-point registers are used. The address register AR stores the address for memory. The data transfer with memory is done through data register DR. The instruction register stores the opcode portion of the instruction. The Program Status Word (PSW) is a two 32-bit words. It stores program status, interrupts that the CPU may respond to and the address of the next instruction to be executed.

1.2.5 Later Generations

Beyond the third generation, it is not easy to define generation of computers. Later generations are based on the advances in integrated-circuit technology. The impact of large scale integration (LSI) and very large scale integration (VLSI) technology on computer design has been profound. It has made it possible to fabricate an entire CPU, main memory or similar device with a single IC that can be mass produced at very low cost. This has resulted in new classes of machines such as personal computers and high-performance parallel processors that contain thousands of CPUs.

Review Questions

1. *Explain the drawbacks of mechanical machines.*
2. *Draw and explain the structure of Von Neumann machine.*
3. *Give the data and instruction formats used by Von Neumann machine.*
4. *List the instructions supported by IAS computer.*
5. *Explain brief history of computers.*
6. *What do you mean by instruction cycle ? Explain the basic instruction cycle with the help of flowchart.*
7. *Explain three phases of instruction cycle with the help of suitable example.*
8. *Briefly explain the organization of IAS computer with it's instruction set.* **Dec.-11, Marks 8**
9. *With a neat diagram explain Von Neumann computer architecture.* **May-07, Marks 12**

1.3 VLSI Era

Dec.-03, 08

As mentioned earlier, the very large scale integration (VLSI) allows manufacturers to fabricate a CPU, main memory or even all the electronic circuits of a computer, on a single IC that can be mass-produced at very low cost. This has resulted in new classes of machines ranging from portable personal computers to supercomputers that contain thousands of CPUs.

1.3.1 Integrated Circuits

Since the 1960s a powerful technology for manufacturing different circuits has been the integrated circuit or IC. An integrated circuit is a group of transistors, diodes, resistors and sometimes capacitors wired together on a very small substrate (or wafer).

Using IC technology, tens of thousands of components can be contained in a single integrated circuit. There are many advantages offered by integrated-circuit technology as compared with discrete components interconnected by conventional techniques.

The important advantages are :

1. Low cost 2. Small size 3. High reliability 4. Improved performance

Large Scale Integration (LSI) is an extension of integrated-circuit (IC) techniques. LSI represents the process of fabricating chips with a large number of components which are interconnected to form complete subsystems or systems. In 1972, typically more than 100 gates or 1000 individual circuit components were contained in commercially available LSI circuits.

Medium Scale Integration (MSI) devices have density of components less than LSI but more than about 100 per chip.

The latter case is Very Large Scale Integration i.e. VLSI. The impact of large scale integration (LSI) and very large scale integration (VLSI) technology on computer design has been profound. The third and later generations of computer are based on the advances in integrated circuit technology. VLSI technology made it possible to fabricate an entire CPU, main memory and IO circuits of a computer on a single IC that can be mass-produced at very low cost. This has resulted in new classes of machines such as personal computers and high-performance parallel processors that contain thousands of CPUs. ICs can be manufactured in high volume at low cost per circuit.

For any computer system, system performance depends on the processor as well as the performance of main memory system. Dynamic Random-Access Memory (DRAM) is a basic building block of main memory. Around 1970, a pocket calculator was manufactured using a single IC chip. After this great achievement, single-chip DRAMs and microprocessors were developed. There is continuous development in the IC technology. Because of this, DRAM chip was 1 K = 2^{10} bits in 1970, has been growing continuously and now a days 1G-bits DRAMs are available. As IC technology improved and chip density (i.e. number of transistors contained in the chip) increased, the complexity and performance of one-chip microprocessors increased steadily. This is reflected in the increase in CPU word size from 4-bit, 8-bit, 16-bit, 32-bit and upto 64-bit. By 1990 it became possible to fabricate entire CPU along with part of its main memory, on a single IC.

1.3.2 IC Families

According to the transistor and circuit types employed, different sub technologies exist within IC technology. The most important among these are, bipolar and MOS (Metal-Oxide-Semiconductor) which is also referred to as unipolar. The basic elements in both the types of circuits are transistors. The difference between them is in the polarities of electric charges associated with primary carriers of electrical signals within their transistors. Bipolar circuits use both the types of carriers whereas MOS circuits use

negative carriers (electrons in case of NMOS) and positive carriers (holes in case of PMOS). CMOS is the MOS family which combines PMOS and NMOS transistors in the same IC. This technology was widely used in 1980s. This is preferred by many manufacturers for microprocessors and other VLSI ICs because of its advantages such as high density, high speed and at the same time very low power consumption.

1.3.3 Processor Architecture

By 1980 computers were classified by their sizes and capabilities as mainframe computers, minicomputers and microcomputers.

Mainframe computers are implemented using two or more central processing units (CPUs). These are designed to work at very high speeds with large data word lengths, typically 64-bits or greater. The data storage capacity of these computers is very high. These are used for complex scientific, large data processing applications, Military defense control and for complex graphics applications.

Minicomputers are the scaled down version of mainframe computer with the moderate speed and storage capacity. These are designed to process smaller data words, typically 32-bit words. This type of computers are used for scientific calculations, research, data processing application etc.

Micro-computers are smaller computers. They contain only one Central Processing Unit CPU (microprocessor) is usually a single integrated circuit. Microcomputer is the integration of microprocessor and supporting peripherals (memory and I/O devices). The word length depends on the microprocessor used and is in the range of 8-bits to 32-bits. This type of computers are used for small industrial control, process control and where storage and speed requirements are moderate.

In the mid 1970s, microcomputer technology gave rise to a new class of machines called personal computers (PCs). A typical PC has the Von Neumann organization. That is, it includes a microprocessor, a multi-megabyte main memory, I/O devices (e.g. keyboard, video monitor or screen), a magnetic or optical disk drive unit for high capacity secondary memory and interface circuits for connecting PC to I/O devices and to other computers. PCs are very much useful in offices and homes for education, entertainment and increasingly, communication with other computers via the World Wide Web (WWW).

In 1981, a personal computer family IBM PC was introduced, which became the most successful PC family. The IBM PC series began with 8086 microprocessor. Because of the advances in VLSI technology, processor hardware became much less expensive and computer designers increased the use of complex, multistep instructions. The 8086 microprocessor was followed by 80186, 80286, 80386, 80486 and pentium. In 1984, Motorola 680×0 microprocessor family was introduced by Apple Computer's Macintosh

which was another popular personal computer series. The advances in VLSI made it possible to add new features such as new instructions, data types, instruction sets and addressing modes to old microprocessors. But at the same time, the ability to execute programs written for older machines was kept constant. A single complex instruction can be used to replace a number of instructions for a given task. For example, multiplication operation can be performed by repeated execution of add instructions. In 8086 microprocessor, multiplication instruction is available which can replace repeated execution of add instructions. This also reduces the overall program execution time. The Intel 80×86/pentium series illustrates the trend toward more complex instruction set. The Intel 8086 microprocessor chip contains 20,000 transistors. It was designed to process 16-bit data words. After twenty-five years, Pentium was introduced by Intel which contained over 3 million transistors. It can process 32-bit and 64-bit words directly. Also it can process floating point operations. It is superscalar processor. i.e. it is capable of parallel instruction execution of multiple instructions.

The 80×86, 680×0 are called **complex instruction set computers** (CISC). The complex instructions reduce program size. But this technology does not translate into faster program execution. However, complex instructions require relatively complex processing circuits. So it becomes necessary to put CISCs in largest and most expensive IC category. This reduces the computer's overall performance. To overcome this drawback, a new processor, **Reduced Instruction Set Computer** (RISC) was introduced by IBM. In 1980s, a number of commercially successful RISC microprocessors were introduced such as IBM RISC System/6000 and SPARC.

Advances in VLSI technology affecting all types of computers tend to increase the CPU's clock frequency and hence to reduce program execution time.

The PowerPC architecture was developed by Motorola, IBM and Apple computer, is based on the POWER architecture implemented by RS/6000^{TM} family of computers. The PowerPC architecture takes advantage of recent technological advances in such areas as process technology, compiler design and RISC microprocessor design to provide software compatibility across a diverse family of implementations, primarily single chip microprocessors, intended for a wide range of systems, including battery-powered personal computers; embedded controllers, high-end scientific and graphics workstations and multiprocessing, microprocessor based mainframes.

Because of further advances in VLSI, Intel came out with the first 8-bit microcontroller, 8048. Unlike microprocessors, microcontrollers are generally optimized for specific applications. The Intel 8048 was designed to control the general task. After that, the high performance microcontroller families such as MCS51, MCS96 were developed. The 8051 in the MCS51 family was optimized for 8-bit math and single bit Boolean operations and the 8096 in the MCS96 family was designed for high speed/high

performance control applications. Overall these families provide larger program and data memory spaces, more flexible I/O and peripheral capabilities, greater speed and lower system cost than any previous generation single chip microcontroller.

The IC technology has also been the driving force in the proliferation of large-scale computer networks-the **Internet**. All this discussion shows us the impact of VLSI on computer design and application.

1.3.4 Performance

When we say one computer is faster than another, we compare their speeds and observes that the faster computer runs a program in less time than other computers. The computer centre manager running a large server system may say a computer is faster when it completes more jobs in an hour. The computer user is always interested in reducing the time between the start and the completion of the program or event, i.e. reducing the **execution time**. The execution time is also referred to as **response time**. Reduction in response time increases the **throughput** (the total amount of work done in a given time). The performance of the computer is directly related to throughput and hence it is reciprocal of execution time.

$$\text{Performance}_A = \frac{1}{\text{Execution time}_A}$$

This means that for two computers A and B if the performance of A is greater than the performance of B, we have,

$$\text{Performance}_A > \text{Performance}_B$$

$$\frac{1}{\text{Execution time}_A} > \frac{1}{\text{Execution time}_B}$$

$$\text{Execution time}_B > \text{Execution time}_A$$

That is, the execution time on B is longer than that on A, if A is faster than B.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "A is n times faster than B" or equivalently "A is n times as fast as B" to mean.

$$\frac{\text{Performance}_A}{\text{Performance}_B} = n$$

If A is n times faster than B then the execution time on B is n times longer than it is on A :

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Example 1.3.1 *If computer A runs a program in 10 seconds and computer B runs the same program in 25 seconds, how much faster is A than B ?*

Solution : We know that A is n times faster than B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is,

$$\frac{25}{10} = 2.5$$

and A is therefore 2.5 times faster than B.

In the above example, we could also say that computer B is 2.5 times slower than computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 2.5$$

means that

$$\frac{\text{Performance}_A}{2.5} = \text{Performance}_B$$

For simplicity, we will normally use the terminology faster than when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms increasing and decreasing, we usually say "improve performance" or "improve execution time" when we mean "increase performance" and "decrease execution time".

The ideal performance of a computer system is achieved when we have a perfect match between the machine capability and the program behaviour. The machine capability can be enhanced with better hardware technology, innovative architectural features and efficient resources management. However, program behaviour is difficult to predict since it heavily depends on application and run-time conditions. The program behaviour also depends on the algorithm design, data structures used, language efficiency, programmer skill and compiler technology. Let us see the factors for projecting the performance of a computer.

1.3.4.1 Processor Clock

In today's digital computer, the CPU or simply the processor is driven by a clock with a constant cycle time called processor clock. The time period of processor clock is denoted by P. The period P of one clock cycle is an important parameter that affects processor performance. The clock rate is given by R = 1/P which is measured in cycles

per second (CPS). The electrical unit for this measurement of CPS is **hertz** (Hz). Today's personal computers and workstations have clock rates in the range of megahertz (MHz) and gigahertz (GHz). The computers having clock rate of 800 MHz have 800 million cycles per second.

1.3.4.2 CPU Time

CPU execution time or simply **CPU time** is the time the CPU spends computing for particular task and does not include time spend waiting for I/O or running other programs. CPU time can be divided into the CPU time spent in the program, called user CPU time and the CPU time spent in the operating system performing tasks on behalf of the program, called system CPU time. Differentiating between system and user CPU time is difficult to do accurately because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems. We use CPU performance to refer user CPU time.

1.3.4.3 Performance Metrics

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as seen by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time :

CPU execution time = CPU clock cycles for a program × Clock cycle time for a program

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program.

Hardware Software Interface

The previous equation do not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program.

For the execution of program, processor has to execute number of machine language instructions. This number is denoted by N. The number N is the actual number of instructions executed by the processor and is not necessarily equal to the number of machine instructions in the machine language program. This is because some instructions may be executed more than once in the loop and others may not be executed at all. Each machine instruction takes one or more cycle time for execution. This time is required to perform various steps needed to execute machine instruction. The average number of basic steps required to execute one machine instruction is denoted by S, where each basic step is completed in one clock cycle. Thus, the program execution time is given by,

$$T = \frac{N \times S}{R} \quad \ldots (1.3.1)$$

where N is the actual number of instructions executed by the processor for execution of a program, R is a clock rate measured in clocks per second and S is the average number of steps needed to execute one machine instruction. The above equation is known as **basic performance equation**.

When machine instruction execution time is measured in terms of **cycles per instruction** (CPI), the program execution time is given as

$$T = \frac{N \times CPI}{R} \quad \ldots (1.3.2)$$

We know that each instruction execution involves cycle of events involving the instruction fetch, decode, operand(s) fetch, execution and store results. We need to access memory to perform instruction fetch, to perform operand(s) fetch or to store results.

The memory cycle is the time needed to complete one memory reference. Usually, a memory cycle is k times the processor cycle, P. The value of k depends on the speed of the memory technology and the interconnection scheme used to interface memory and processor.

The CPI of the an instruction type can be divided into two components terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction. Therefore, we can rewrite the equation 2 as,

$$T = \frac{N \times (p + m \times k)}{R} \quad \ldots (1.3.3)$$

Where p is the number of processor cycles required for the instruction decode and execute, m is the number of memory references needed, k is the ratio between memory cycle and processor cycle, N is the machine instruction count and R is the clock rate. The above performance parameters, i.e. N, p, m, k, R are affected by four system

attributes : instruction set architecture, compiler technology, CPU implementation and control and cache and memory hierarchy, as shown Table 1.3.1.

System attributes	Performance parameters				
	Machine instruction count (N)	Average cycles per instruction			Clock rate (R)
		Processor cycles per instruction (p)	Memory references per instruction (m)	Memory access latency, k	
Instruction set architecture	✓	✓			
Compiler technology	✓	✓	✓		
Processor implementation and control		✓			✓
Cache and memory hierarchy				✓	✓

Table 1.3.1

The instruction set architecture affects the machine instruction count (N), i.e. the program length and the average processor cycles required per instruction (p). The compiler technology affects the value of N, p and the memory reference count (m). The processor implementation and control determine the total processor time (p/R) required. Finally, the memory technology and hierarchy design affect the memory access latency (k/R).

Example 1.3.2 *Let us assume that two computers use same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much ?*

Solution : We know that each computer executes the same number of instructions for the program; let's call this number N. First, find the number of processor clock cycles for each computer :

$$\text{CPU clock cycles}_A = N \times 2.0$$

$$\text{CPU clock cycles}_B = N \times 1.2$$

The CPU time for each machine will be

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}_A$$

$$= N \times 2.0 \times 250 \text{ ps} = 500\,N \text{ ps}$$

$$\text{CPU time}_B = \text{CPU clock cycles}_B \times \text{Clock cycle time}_B$$

$$= N \times 1.2 \times 500 \text{ ps} = 600\,N \text{ ps}$$

Thus we can say that computer A is faster. The amount faster is given by the ratio of the execution times.

$$\frac{\text{CPU Performance}_A}{\text{CPU Performance}_B} = \frac{\text{Execution times}_B}{\text{Execution times}_A} = \frac{600 \text{ N ps}}{500 \text{ N ps}} = 1.2$$

We can conclude that computer A is 1.2 times faster than computer B for this program.

Other Performance Measures

MIPS is an another way to measure the processor speed. The processor speed can be measured in terms of **million instructions per second** (MIPS). It is given as,

$$\text{MIPS rate} = \frac{1}{\text{Average time required for the execution of instruction} \times 10^6}$$

$$= \frac{R}{CPI \times 10^6} = \frac{N \times R}{N \times CPI \times 10^6} \quad \ldots (1.3.4)$$

Substituting value of T from equation (1.3.2) we get,

$$\text{MIPS rate} = \frac{N}{T \times 10^6} \quad \ldots (1.3.5)$$

Referring equation (1.3.2) we can also write

$$\text{MIPS rate} = \frac{N \times R}{C \times 10^6}$$

Where C is the total number of clock cycles required to execute a given program (N× CPI).

Throughput Rate

Another important measure of throughput is known as **throughput rate**. It indicates a number of programs a system can execute per unit time. It is often specified as programs/second. Throughput can be further measured separately for the system (W_s) and for the processor (W_p). The processor throughput is given as,

$$W_p = \frac{\text{Number of machine instructions executed per second}}{\text{Number of machine instructions per program}} \quad \ldots (1.3.6)$$

$$= \frac{\text{MIPS rate} \times 10^6}{N}$$

It is often greater than the system throughput because in system throughput we have to consider the system overheads caused by the I/O, compiler and OS (operating system) when multiple programs are interleaved for processor execution by multiprogramming or time sharing. If the processor is kept busy in a perfect program

interleaving fashion, then $W_s = W_p$. This will probably never happen, since the system overhead often causes an extra delay and the processor may be left idle for some cycles.

MFLOPS

The 1970s and 1980s marked the growth of the supercomputer industry which was defined by high performance on floating-point-intensive programs. Average instruction time and MIPS were clearly inappropriate metrics for this industry. Hence another popular alternative to execution time was invented. It is million floating-point operations per second, abbreviated **megaflops** or **MFLOPS** but always pronounced "megaflops." MFLOPS can be defined as

$$\text{MFLOPS} = \frac{\text{Number of floating point operations in a program}}{\text{Execution time} \times 10^6}$$

Where, a floating-point operation is an addition, subtraction, multiplication or division operation applied to a number in a single or double precision floating point representation. Such data items are heavily used in scientific calculations and are specified in programming languages using key words like float, real, double or double precision.

MFLOP rating is dependent on the program. Different programs require the execution of different number of floating point operations. Since MFLOPS were intended to measure floating-point performance, they are not applicable outside that range.

MFLOPS is based on operations in the program rather than on instructions, hence it has a stronger claim than MIPS to being a fair comparison between different machines. The key point is that the same program running on different computers may execute a different number of instructions but will always execute the same number of floating-point operations. Unfortunately, MFLOPS is not dependable because the set of floating-point operations is not consistent across machines and the number of actual floating-point operations performed may vary. For example, the processor which does not provide division instruction requires several floating-point operations to perform floating-point division, whereas the processor which provides division instruction requires only one floating point operation to perform floating point division.

Another major problem is that the MFLOPS ratings changes according not only to the mixture of integer and floating-point operations but to the mixture of fast and slow floating-point operations. For example, the program with floating-point add operations have a higher rating than the program with floating-point division operations. This problem can be solved by giving more weights to the complex floating-point operations while measuring the performance. These MFLOPS might be called **normalized MFLOPS**. Of course, because of the counting and weighting, these normalized MFLOPS may be very different from the actual rate at which a machine executes floating-point operations.

1.3.4.4 Performance Measurement

When we compare the performance of different computers say A, B and C, we may observe that some programs run faster by computer A, some by computer B and some by computer C. In this situation they present a confusing picture and we cannot have a clear idea of which computer is faster. This happens because each computer has an ability to execute particular instruction or step in the instruction execution faster than others.

We know that, processing of an instruction involves several steps :

- Fetch the instruction from main memory M.
- Decode the instruction opcode.
- Load the operands from the main memory if they are not in the CPU registers.
- Execute the instruction using appropriate functional unit, such as floating point adder or fixed point adder.
- Store the results in the main memory unless they are to be retained in CPU registers.

All instructions do not require to perform all steps listed above. When instruction has all its operands in CPU registers, it will run faster whereas the instruction which requires multiple memory accesses takes more time to execute. Let us consider two programs P_1 and P_2, with instructions having all operands in the CPU and with instructions having all operands in the memory, respectively. Also consider two computers C_1 and C_2. The clock speed of C_1 is greater than the clock speed of C_2 ; however the memory access time in C_1 is less than the memory access time in C_2. With these computer conditions we can easily understand that the C_1 will execute the program P_1 faster than C_2 and C_2 will execute the program P_2 faster than C_1. In such situation it is difficult to decide which computer is faster. Therefore, measures of instruction execution performance are based on average figures, which are usually determined experimentally by measuring the run times of representative called **benchmark programs**. In recent years, it has become popular to put together collection of benchmarks to try to measure the performance of processors with a variety of applications. The benchmark programs are different for checking the performance of processor for different applications. According to applications the benchmark programs are classified as :

- Desktop benchmark
- Server benchmark and
- Embedded benchmark

Desktop Benchmarks

Desktop benchmarks divide into two broad classes : **CPU-intensive benchmarks** and **graphics-intensive benchmarks**. These two benchmark programs measures compute CPU and graphics performance of the processor, respectively.

Server Benchmarks

We know that servers have to perform many functions, so there are multiple types of benchmark programs for servers.

- **CPU throughput oriented benchmark :** This benchmark program can be used to measure the processing rate of a multiprocessor by running multiple copies, one for each CPU benchmark and converting the CPU time into a rate. This particular measurement is known as **SPEC rate**.

- **Web server benchmark :** This benchmark program simulates multiple clients requesting both static and dynamic pages from a server, as well as clients posting data to the server.

- **File system benchmark :** It is used to measure network file system (NFS) performance using a script of file server requests. It also tests the performance of the I/O system (both disk and network I/O) as well as the CPU.

- **Transaction processing benchmark :** It is used to measure the ability of a system to handle transactions, which consists of database accesses and updates. In the mid 1980s, a group of concerned engineers formed the vendor-independent transaction processing council (TPC) to try to create a set of realistic and fair benchmark programs for transaction processing. Following this TPC benchmark program there were many benchmarks published namely TPC-A, TPC-C, TPC-H, TPC-R and TPC-W. All these benchmarks, measure performance in transactions per second. In addition, they include a response time requirement, so that throughput performance is measured only when the response time limit is met.

Embedded Benchmarks

Embedded applications have enormous variety and their performance requirements are also different. Thus, it is unrealistic to have a single set of benchmark programs for embedded systems. In practice, many designers of embedded systems planned for the benchmark programs that reflect their application, either as kernels or as stand-alone versions of the entire application.

A new set of standardized benchmark programs (EDN) Embedded Microprocessor Benchmark Consortium are available for embedded applications which are characterized well by Kernel performance. These benchmark programs are divided into five different classes :

- Automotive/industrial
- Consumer
- Networking
- Office automation
- Telecommunications

Automotive/industrial benchmark programs include microbenchmark programs for arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup and bit manipulation. They also include automobile control benchmarks and FFT benchmarks. The consumer benchmark programs include mainly multimedia benchmarks like JPEG compress/decompress, filtering and RGB conversions. Networking benchmark is the collection of programs for shortest path calculations, IP routing and packet flow operations. Office automation benchmark programs include graphics and text benchmarks such as Bezier curve calculation, dithering, image rotation and text processing. Finally, telecommunication benchmark programs include filtering and DSP benchmarks.

The selected benchmark programs are compiled for the computer under test and the running time on a real computer is measured. The same benchmark program is also compiled and run on the reference computer. A nonprofit organization called System Performance Evaluation Corporation (SPEC) specified the benchmark programs and reference computers in 1995 and again in 2000. For SPEC95, the reference computer is the SUN SPARCStation 10/40 and for SPEC2000, the reference computer is an Ultra-SPARC10 workstation with a 300 MHz Ultra SPARC-II processor.

The running time of a benchmark program is compared for computer under test and the reference computer to decide the SPEC rating of the computer under test. The SPEC rating is given by

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

The SPEC rating for all selected programs is individually calculated and then the geometric mean of the results is computed to determine the overall SPEC rating for the computer under test. It is given by,

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i \right)^{\frac{1}{n}}$$

where n is the number benchmark programs used for determining SPEC rating. The computers providing higher performance have higher SPEC rating.

Review Questions

1. *Write a short note on VLSI Era.*
2. *Explain the advantages offered by integrated circuit technology.*
3. *Write a short note on IC family.*
4. *Write a short note on evolutions of computers and their performance considerations.*
5. *What are the important factors that determine a computer's performance ? Give its significance.*

Dec.-03, Marks 2

6. *Write the basic performance equation and using this equation explain how the performance of a system can be improved.* **Dec.-08, Marks 16**

1.4 Design

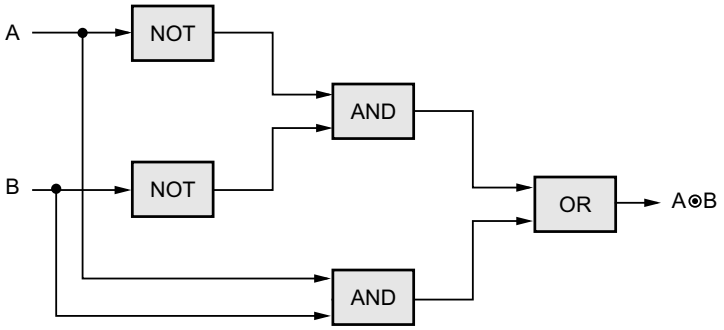
This section explains the design process for digital system at two basic levels of abstraction : the register level and processor level.

1.4.1 System Design

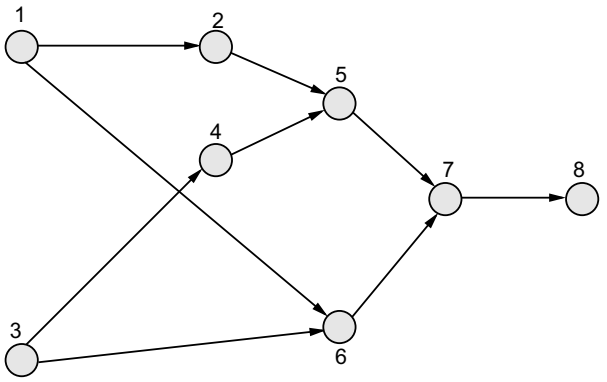
A computer is a large and complex system in which system objects are the components of the computer. These components are connected to perform a specific function. The function of such system is determined by the functions of its components and how the components are connected.

1.4.1.1 System Representation

We can represent a system using a graph or a block diagram. A computer system is usually represented by a block diagram. A system has its own structure and behaviour. The structure and behaviour are the two properties of the system. We can define the structure of a system as the abstract graph consisting of its block diagram with no functional information, as shown in Fig. 1.4.1.



(a) A block diagram representing EX-NOR logic circuit



(b) Structure of a system as an abstract graph

Fig. 1.4.1

As shown in Fig. 1.4.1, the structure gives the components and their interconnection. A behavioral description, on the other hand, describes the function of each component and thus the function of the system. The behaviour of the system may be represented by Boolean function or by truth table in case of logic circuit.

The behaviour of logic circuits can also be described by Hardware description language such as VHDL. They can provide precise, technology-independent descriptions of digital circuits at various levels of abstraction, primarily the gate and register levels.

1.4.1.2 Design Process

For a given system's structure, the task of determining its function or behaviour is termed **analysis**. On the other hand, the problem of determining a system structure that exhibits a given behaviour is **design** or **synthesis**.

The design process starts with the construction of initial design. In this process, with given a desired range of behaviour and set of available components we have to determine a structure (Design).

The next step is to evaluate its cost and performance. The cost and performance should be in the acceptable range. Then we have to confirm that whether the formed structure achieves the desired behaviour. If not we have to modify the design to meet the design goals. The Fig. 1.4.2 illustrates the design process.

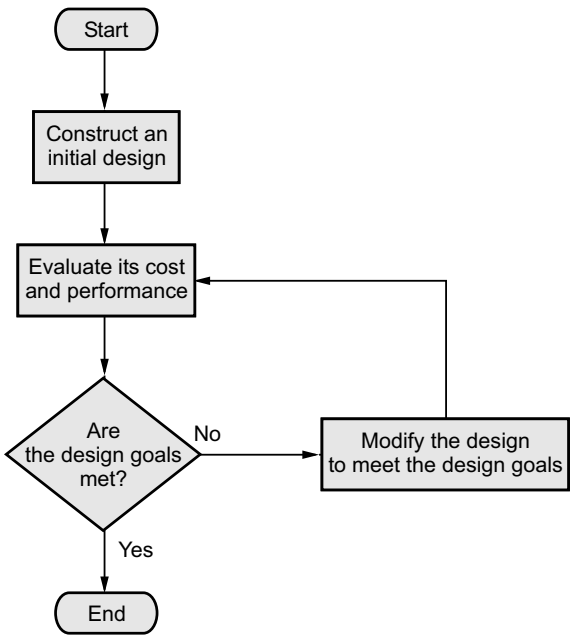


Fig. 1.4.2 Design process

Computer-aided Design

The computer-aided design (CAD) tools provides designers with a range of programs to support their design goals. They are used to automate fully or party the more tedious design and evaluate its steps. They contribute mainly in three important ways to the overall design process.

- CAD editors or translators convert design data into forms such as HDL descriptions or schematic diagrams, which can be efficiently processed by the humans, computers or both.
- Simulators creates the computer model for the design and can mimic the design's behaviour. It helps designer to determine how well the design meets various performance and cost goals.
- Synthesizers derives structures that implement all or part of some design step.

Design Levels

The design of a computer system can be carried at several levels of abstraction. The three such recommended levels are :

- The processor level also called the architecture, behaviour or system level.
- The register level, also called the register-transfer level (RTL).
- The gate level, also called the logic level.

The Table 1.4.1 shows the comparison between these levels.

Design Level	Components	IC Density	Information Units	Time Units
Processor	CPUs, memories, IO devices.	VLSI	Blocks of words	10^{-3} to 10^{3} s
Register	Registers, counters, combinational circuits, small sequential circuits.	MSI	Words	10^{-9} to 10^{-6} s
Gate	Logic gates, flip-flops	SSI	Bits	10^{-12} to 10^{-9} s

Table 1.4.1 Comparison between design levels

To design a complex system usually we have to

- Specify the processor-level structure of the system.
- Specify the register-level structure of each component type identified in step 1.
- Specify the gate-level structure of each component type identified in step 2.

This design approach is known as **top-down** design approach and it is extensively used in both hardware and software designs. Well, it is up to the designer to decide whether to design a system using medium scale ICs, small scale ICs or a single IC composed of standard cells. If the system is to be designed using medium-scale ICs or standard cells, then the third step, gate-level design, is no longer needed. In the following section we discuss the register-level and processor-level design approaches.

Review Questions

1. *Explain the design process for a digital system.*
2. *What do you understand by design levels in the design of computer system ?*
3. *Explain top-down design approach.*

1.5 Register Level

Dec.-03, 11, May-11

At the register or register transfer level, related information bits are grouped to form words or vectors. These words are processed by small combinational or sequential circuits.

1.5.1 Register Level Components

The Table 1.5.1 shows the commonly used register level components and their functions. These components are link to form circuits.

Type	Component	Function
Combinational	Word gates.	Boolean operations.
	Multiplexers and Demultiplexers.	Data routing; general combinational functions.
	Decoders and encoders.	Code checking and conversion.
	Adders and subtractors.	Addition and subtraction.
	Arithmetic logic units.	Numerical and logical operations.
	Programmable logic devices.	General combinational functions.
Sequential	Registers	Information storage
	Shift register	Information storage; serial parallel conversion.
	Counters	Control/timing signal generation.
	Programmable logic devices.	General sequential functions.

Table 1.5.1 Commonly used register level components

The Fig. 1.5.1 shows the generic block representation of a register-level component. The "/m" on the input lines indicate it is a m-bit input bus. A slash '/' with number or letter next to it indicates the multi-bit bus. A bubble on the start or end of the line indicates an active low signal; otherwise it is an active high signal. The input and output data lines are shown separately. Similarly the input and output control lines are also shown separately. The input control lines associated with a multifunction block fall into two broad categories : select lines and enable lines. The select lines specify one of

several possible operations that the unit is to perform and enable lines specify the time or condition for a selected operation to be performed. The output control signals, if any, indicate when or how the unit completes its processing.

Let us see the major combinational and sequential components used in design at the register level.

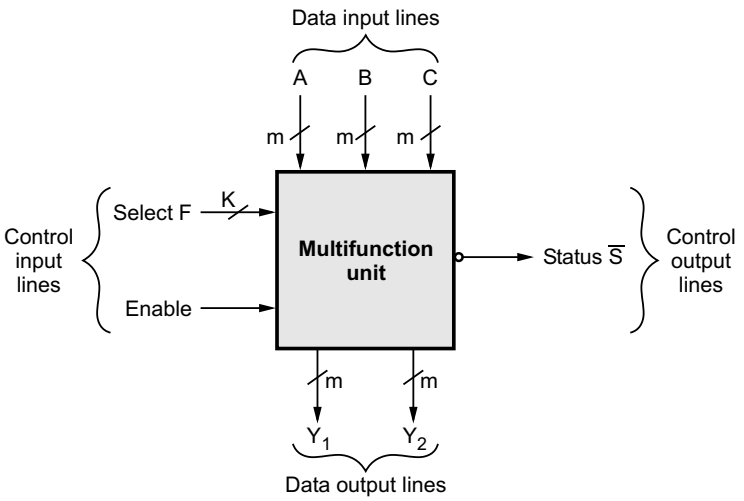
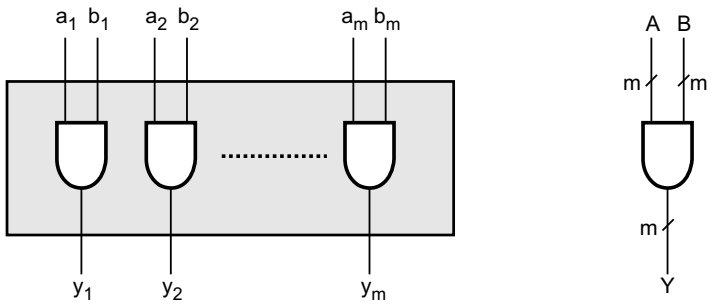


Fig. 1.5.1 Generic block representation of a register-level component

Word Gate : Bit-wise logical functions can be performed on the m-bit binary words using word gate operators. Let $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_m)$ be the two m-bit words we can perform the bitwise AND operation on them to result another m-bit result, as shown in the Fig. 1.5.2.



(a) Logic diagram **(b) Symbol**

Fig. 1.5.2 Two-input, m-bit AND word gate

Multiplexers :

Multiplexer is a digital switch. It allows digital information from several sources to be routed onto a single output line, as shown in the Fig. 1.5.3. The basic multiplexer has several data-input lines and a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected. Therefore, multiplexer is 'many into one' and it provides the digital equivalent of an analog selector switch.

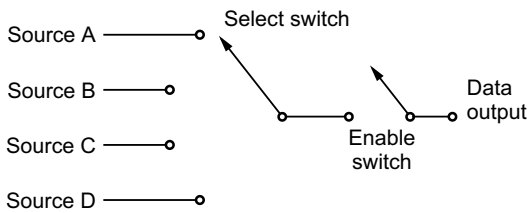
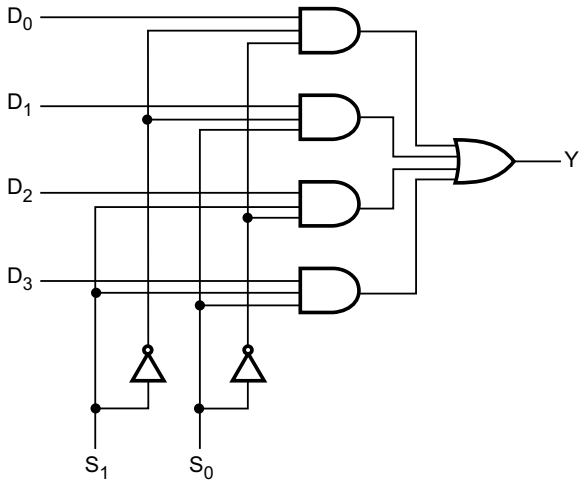


Fig. 1.5.3 Analog selector switch

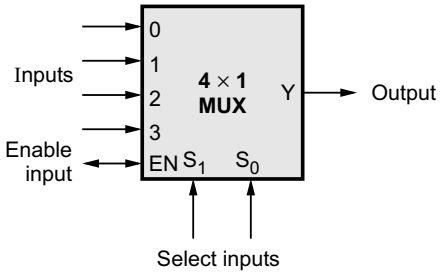
Fig. 1.5.4 shows 4-to-1 line multiplexer. Each of the four lines, D_0 to D_3, is applied to one input of an AND gate. Selection lines are decoded to select a particular AND gate.



(a) Logic diagram

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b) Function table



(c) Logical symbol

Fig. 1.5.4 4-to-1 line multiplexer

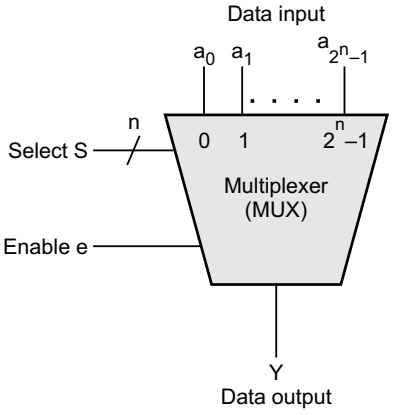


Fig. 1.5.5

For example, when S_1S_0 = 0 1, the AND gate associated with data input D_1 has two of its inputs equal to 1 and the third input connected to D_1. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of D_1, thus we can say data bit D_1 is routed to the output when $S_1 S_0$ = 0 1.

In general, if multiplexer has 2^n inputs then we can represent multiplexer, as shown in Fig. 1.5.5.

In some cases, two or more multiplexers are enclosed within one IC package, as shown in the Fig. 1.5.6. The Fig. 1.5.6 shows quadruple 2-to-1 line multiplexer, i.e. four multiplexers, each capable of selecting one of two input lines. Output Y_1 can be selected to be equal to either A_1 or B_1. Similarly output Y_2 may have the value of A_2 or B_2 and so on. The selection line S selects one of two lines in all four multiplexers. The control input E enables the multiplexers in the 0 state and disables them in the 1 state. When E = 1, outputs have all 0's, regardless of the value of S.

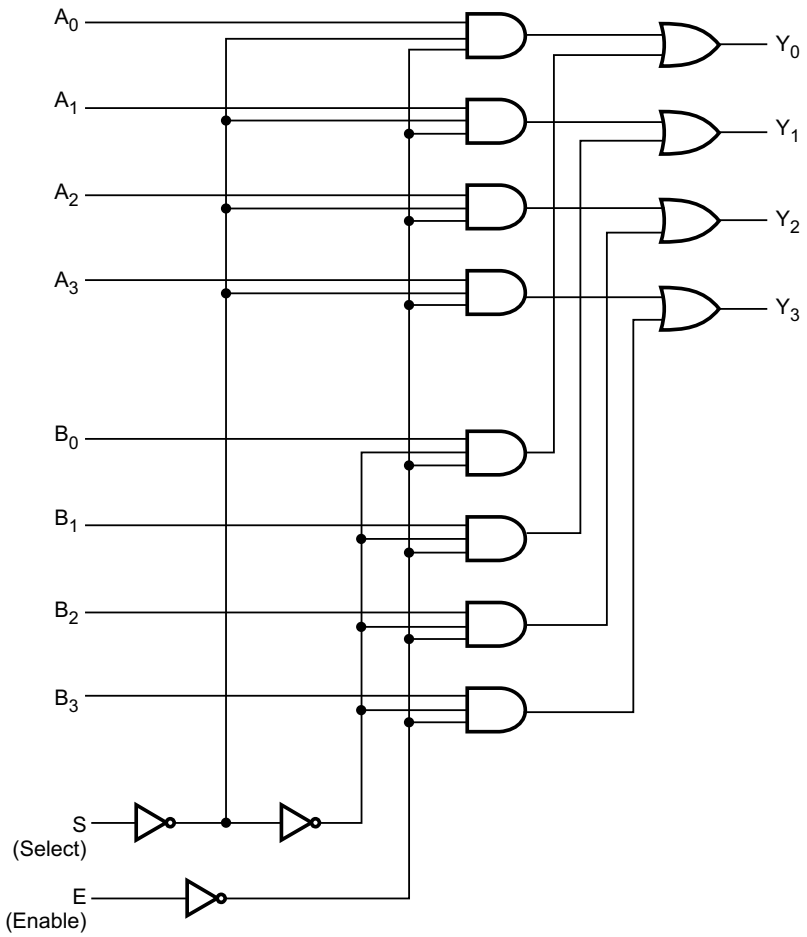


Fig. 1.5.6 Quadruple 2-to-1 line multiplexer

In general, if multiplexer has n multiplexers enclosed with each capable of selecting one of the m input lines, we can represent multiplexer as shown in Fig. 1.5.7. Such a multiplexer is called 2^n-input, m-bit multiplexer.

Expanding Multiplexers

Several digital multiplexer ICs are available such as 74150 (16 to 1), 74151 (8 to 1), 74157 (Dual 2 input) and 74153 (Dual 4 to 1) multiplexer. It is possible to expand the range of inputs for multiplexer beyond the available range in the integrated circuits. This can be accomplished by interconnecting several multiplexers. For example, two 74XX151, 8 to 1 multiplexers can be used together to form a 16-to-1 multiplexer, two 74XX150, 16 to 1 multiplexers can be used together to form a 32 to 1 multiplexer and so on. The Fig. 1.5.8 shows an eight-input multiplexer constructed from two-input multiplexers.

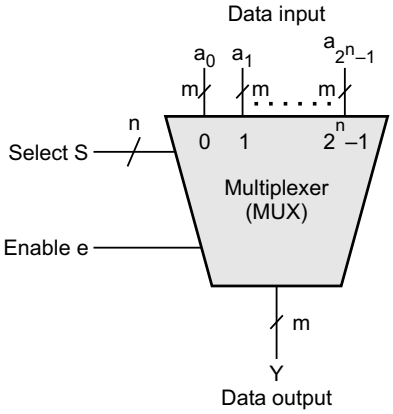


Fig. 1.5.7

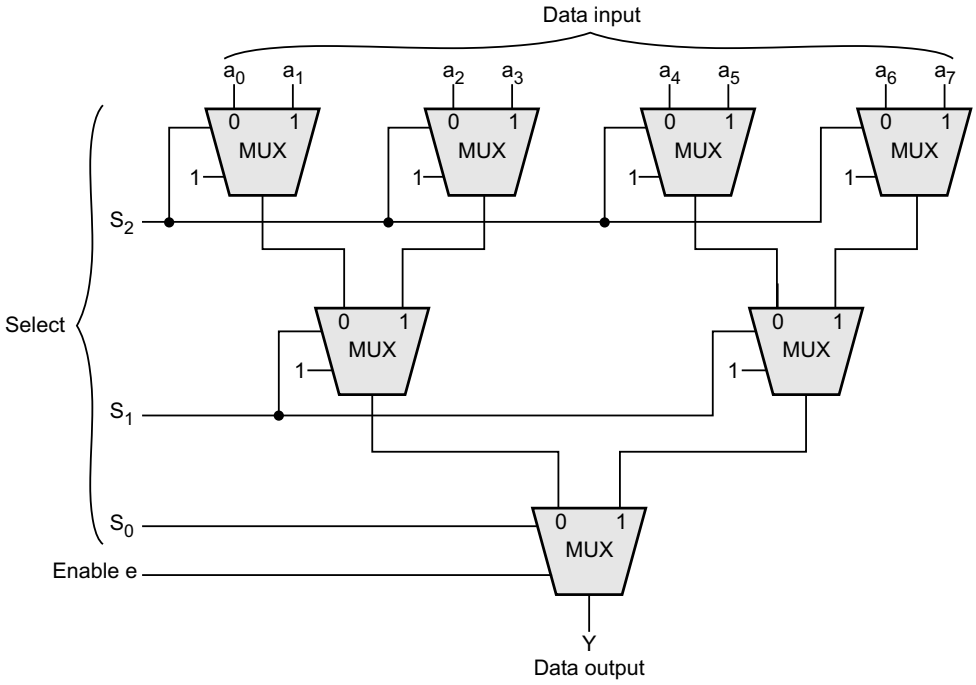


Fig. 1.5.8 An eight-input multiplexer constructed from two-input multiplexers

Multiplexer as Function Generator

A multiplexer consists of a set of AND gates whose outputs are connected to single OR gate. Because of this construction any Boolean function in a SOP form can be easily realized using multiplexer. Each AND gate in the multiplexer represents a minterm. In 8 to 1 multiplexer, there are 3 select inputs and 23 minterms. By connecting the function variables directly to the select inputs, a multiplexer can be made to select the AND gate that corresponds to the minterm in the function. If a minterm exists in a function, we have to connect the AND gate data input to logic 1; otherwise we have to connect it to logic 0. This is illustrated in the following example.

Example 1.5.1 *Implement the following Boolean function using 8 : 1 multiplexer.*
$F(A, B, C) = \sum m(1, 3, 5, 6)$

Solution : The function can be implemented with a 8-to-1 multiplexer, as shown in Fig. 1.5.9. Three variables A, B and C are applied to the select lines. The minterms to be included (1, 3, 5 and 6) are chosen by making their corresponding input lines equal to 1. Minterms 0, 2, 4 and 7 are not included by making their input lines equal to 0.

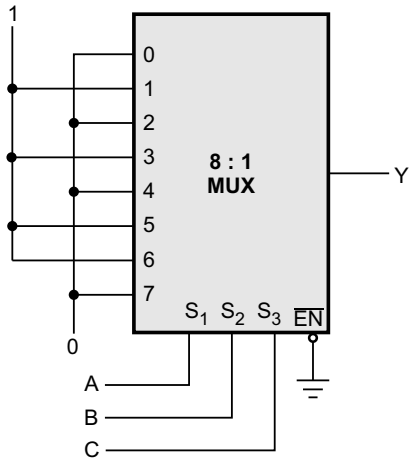


Fig. 1.5.9 Boolean function implementation using MUX

In the above example we have seen the method for implementing Boolean function of 3 variables with 2^3 (8) - to -1 multiplexer. Similarly, we can implement any Boolean function of n variables with 2^n-to-1 multiplexer. However, it is possible to do better than this. If we have Boolean function of n + 1 variables, we take n of these variables and connect them to the selection lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. In this way we can implement any Boolean function of n variables with 2^{n-1}- to-1 multiplexer. Let us see one example.

Example 1.5.2 *What are the major components types at the register level? Using a multiplexer implement a full adder.* **Dec.-03, Marks 8**

Solution : $\text{Carry} = AB + AC_{in} + BC_{in} = \sum m(3, 5, 6, 7)$

$\text{Sum} = \overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C}_{in} + A\overline{B}\,\overline{C}_{in} + ABC_{in} = \sum m(1, 2, 4, 7)$

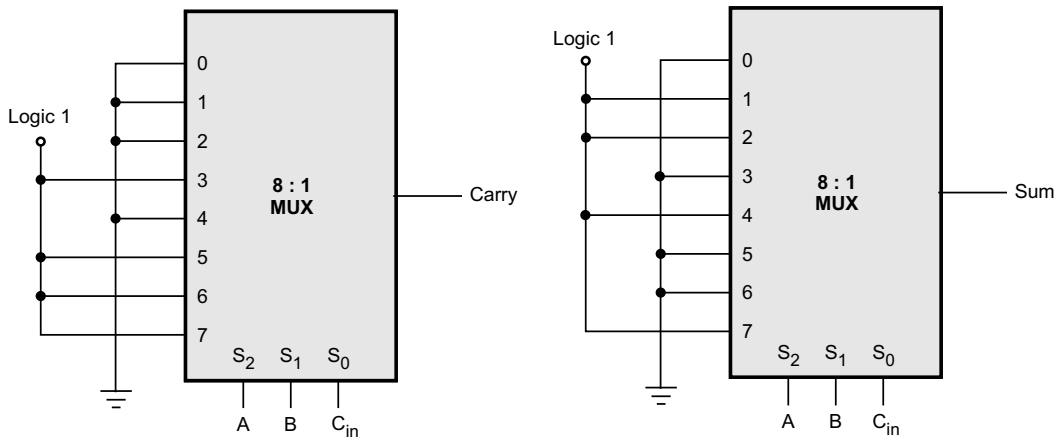


Fig. 1.5.10

Decoder

A decoder is a multiple-input, multiple-output logic circuit which converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code. Each input code word produces a different output code word, i.e., there is one-to-one mapping from input code words into output code words. This one-to-one mapping can be expressed in a truth table.

The Fig. 1.5.11 shows the general structure of the decoder circuit. As shown in the Fig. 1.5.11, the encoded information is presented as n inputs producing 2^n possible outputs. The 2^n output values are from 0 through $2^n - 1$. Sometimes an n-bit binary code is truncated to represent fewer output values than 2^n. For example, in the BCD code, the 4-bit combinations 0000 through 1001 represent the decimal digits 0-9 and combinations 1010 through 1111 are not used. Usually, a decoder is provided with enable inputs to activate decoded output based on data inputs. When any one enable input is unasserted, all outputs of decoder are disabled.

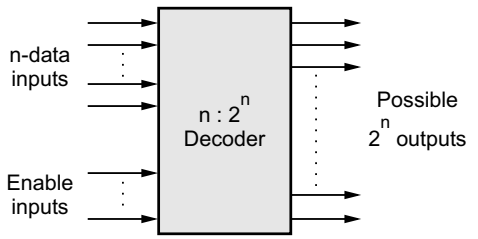


Fig. 1.5.11 General structure of decoder

Encoder

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. In encoder the output lines generate the binary code corresponding to the input value. The Fig. 1.5.12 shows the general

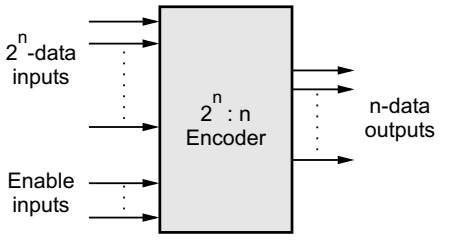


Fig. 1.5.12 General structure of encoder

structure of the encoder circuit. As shown in the Fig. 1.5.12, the decoded information is presented as 2^n inputs producing n possible outputs.

A priority encoder is an encoder circuit that includes the priority function. In priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

Table 1.5.2 shows truth table of 4-bit priority encoder.

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_0	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Table 1.5.2 Truth table of 4-bit priority encoder

Table 1.5.2 shows D_3 input with highest priority and D_0 input with lowest priority. When D_3 input is high, regardless of other inputs output is 1 1. The D_2 has the next priority. Thus, when D_3 = 0 and D_2 = 1, regardless of other two lower priority input, output is 10. The output for D_1 is generated only if higher priority inputs are 0 and so on. The output V (a valid output indicator) indicates, one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0, and the other two outputs (Y_1 and Y_0) of the circuit are not used.

Cascading priority encoders

By cascade connection of several priority encoders, we can obtain a larger priority encoder. In encoder IC, there are two enable signals EI and EO. EI (Input Enable) signal enables the priority encoder while EO (Output Enable) is asserted only when EI is asserted and none of the other inputs are asserted. Thus EO signal can be used to enable other lower priority encoders. The Fig. 1.5.11 shows the cascade connection of two 8-bit priority encoder ICs(74148) to form a 16-bit priority encoder. As shown in the Fig. 1.5.11, EI input of the IC_1 is grounded. If any input of IC_1 goes low, its EO output goes high and disables IC_2 (lower 8-bit priority encoder). The GS output of encoder ICs goes low when any of its input becomes low. The output from both the ICs are again encoded using AND gates. The GS output of IC_1 is used as a most significant bit in the encoded code. The Table 1.5.1 shows the truth table for 16-bit priority encoder.

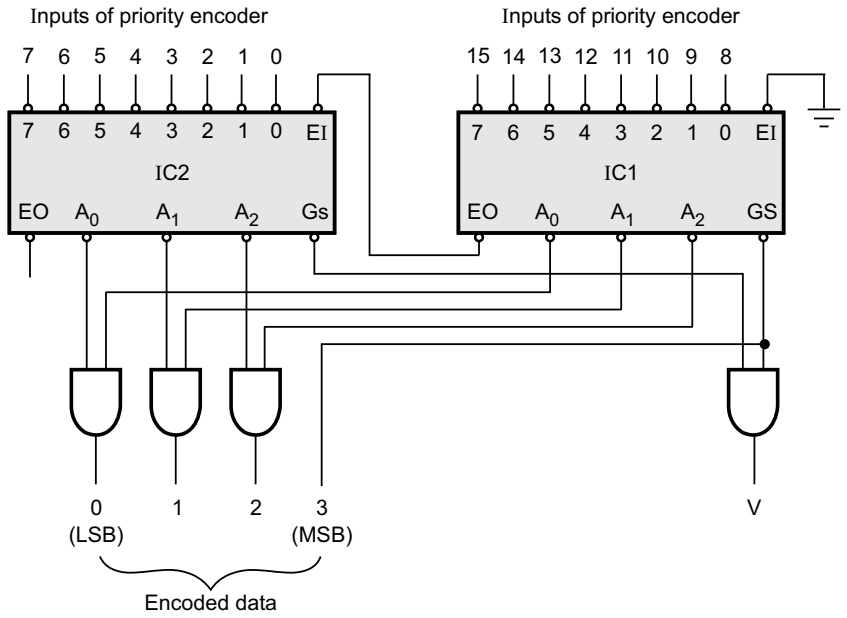


Fig. 1.5.13 16-bit priority encoder

Inputs																Outputs				V
$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$	$\overline{8}$	$\overline{9}$	$\overline{10}$	$\overline{11}$	$\overline{12}$	$\overline{13}$	$\overline{14}$	$\overline{15}$	0	1	2	3	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
×	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0
×	×	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
×	×	×	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0
×	×	×	×	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
×	×	×	×	×	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0
×	×	×	×	×	×	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0
×	×	×	×	×	×	×	0	1	1	1	1	1	1	1	1	0	0	0	1	0
×	×	×	×	×	×	×	×	0	1	1	1	1	1	1	1	1	1	1	0	0
×	×	×	×	×	×	×	×	×	0	1	1	1	1	1	1	0	1	1	0	0

×	×	×	×	×	×	×	×	×	×	0	1	1	1	1	1	1	0	1	0	0
×	×	×	×	×	×	×	×	×	×	×	0	1	1	1	1	0	0	1	0	0
×	×	×	×	×	×	×	×	×	×	×	×	0	1	1	1	1	1	0	0	0
×	×	×	×	×	×	×	×	×	×	×	×	×	0	1	1	0	1	0	0	0
×	×	×	×	×	×	×	×	×	×	×	×	×	×	0	1	1	0	0	0	0
×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	0	0	0	0	0	0

Table 1.5.3 Truth table for 16-bit priority encoder

Demultiplexer

A demultiplexer is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of specific output line is controlled by the values of n selection lines. Fig. 1.5.14 shows 1 : 4 demultiplexer. The single input variable D_{in} has a path to all four outputs, but the input information is directed to only one of the output lines.

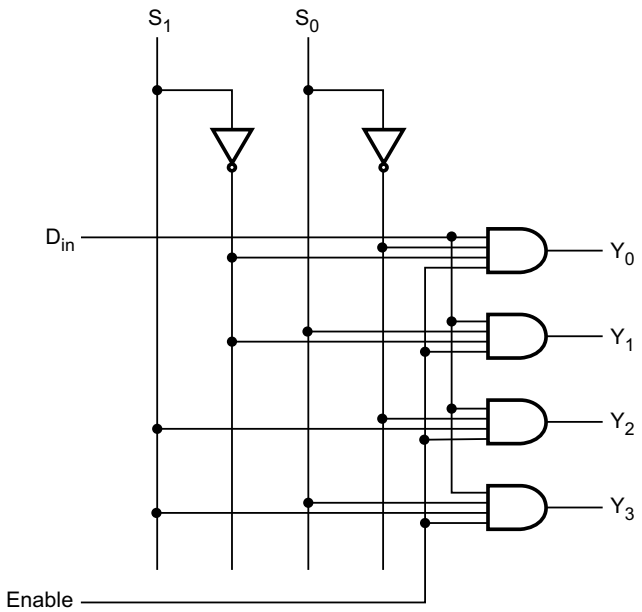


Fig. 1.5.14 (a) Logic diagram

Enable	S_1	S_0	D_{in}	Y_0	Y_1	Y_2	Y_3
0	X	X	X	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	1

Table 1.5.4 Function table for 1:4 demultiplexer

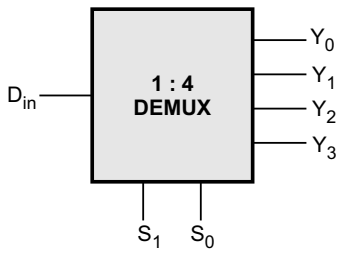
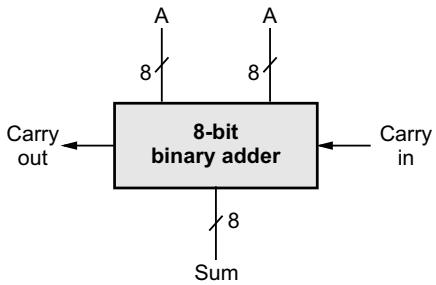


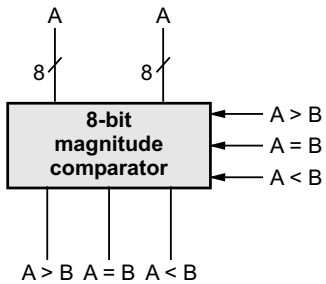
Fig. 1.5.14 (b) Block diagram

Arithmetic Elements

The arithmetic functions such as addition and subtraction of fixed point numbers can be implemented by combinational register-level components. Most forms of fixed-point multiplication and division and essentially all floating-point operations are too complex to be realized by single component at this design level. However, adders and subtractors for fixed-point binary numbers are basic register level components from which we can derive a variety of other arithmetic circuits. The Fig. 1.5.14 (a) shows a component that adds two 8-bit data words and an input carry bit; it is called a 8-bit adder. One such component can be cascaded to form an adder to add numbers of arbitrary size. However, addition time increases with the number size.



(a) Symbol for 8-bit binary adder



(b) Symbol for 8-bit magnitude comparator

Fig. 1.5.15

The magnitude comparator is an another useful arithmetic component, whose function is to compare the magnitudes of two binary numbers. The Fig. 1.5.15 (b) shows the symbol for 8-bit magnitude comparator.

Let us see the design of 8-bit magnitude comparator at register level. To check the number A is greater than number B (A > B) we have to perform following steps :

- Compute $\overline{B}$ from B using an n-bit word inverter.
- Add A and $\overline{B}$ using an n-bit adder and use the output-carry signal C_{out} as the primary output. If $C_{out} = 1$, then A > B ; if $C_{out} = 0$, then A ≤ B.

For example, if A = 10001100 and B = 01001000 then $\overline{B}$ = 10110111 and $A + \overline{B} = 1\ 0100\ 0011$, i.e. $C_{out} = 1$ and A > B

Using similar technique and changing positions of A and B we can derive the output of A < B. To get the 'equals' output we can use Ex-NOR word gate. The Ex-NOR word gate bit-wise compares inputs A and B and gives output word of all 1s if both words are equal. To generate single bit output A = B we have to AND gate word. The Fig. 1.5.16 shows the implementation of above discussed circuit.

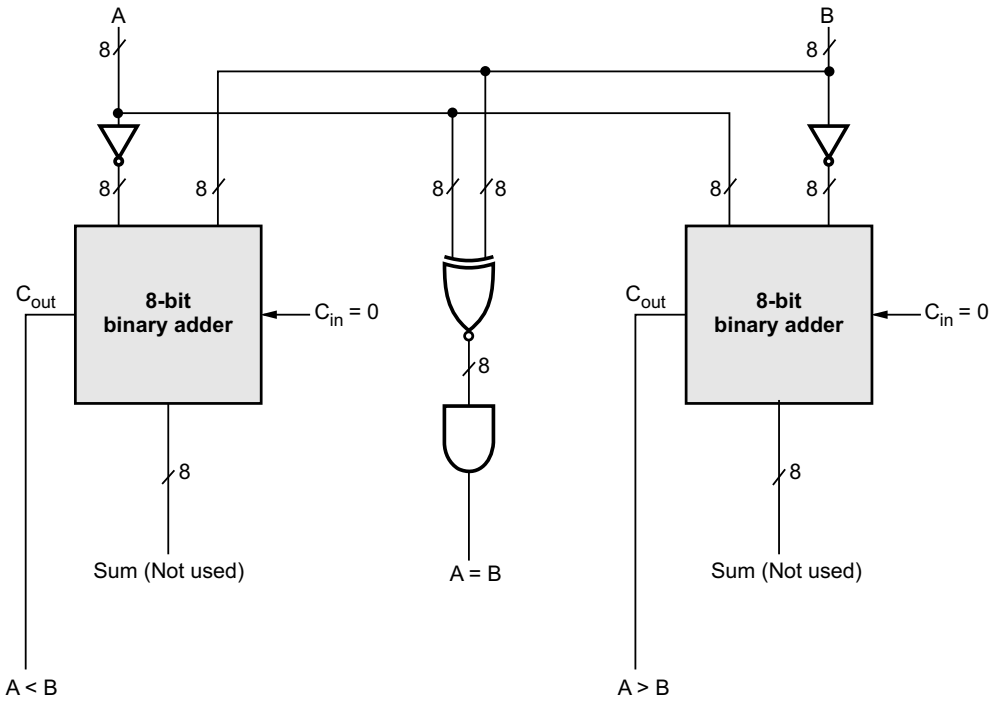


Fig. 1.5.16 Register-level design of a 8-bit magnitude comparator

Register

A register is a group of flip-flops. A flip-flop can store 1-bit information. So an n-bit register has a group of n flip-flops and is capable of storing any binary information/number containing n-bits.

Buffer Register

Fig. 1.5.17 shows the simplest register constructed with four D flip-flops. This register is also called **buffer register**. Each D-flip-flop is triggered with a common negative edge clock pulse. The input X bits set up the flip-flops for loading. Therefore, when the first negative clock edge arrives, the stored binary information becomes,

$$Q_A Q_B Q_C Q_D = ABCD$$

In this register, four D flip-flops are used. So it can store 4-bit binary information. Thus the number of flip-flop stages in a register determines its total storage capacity.

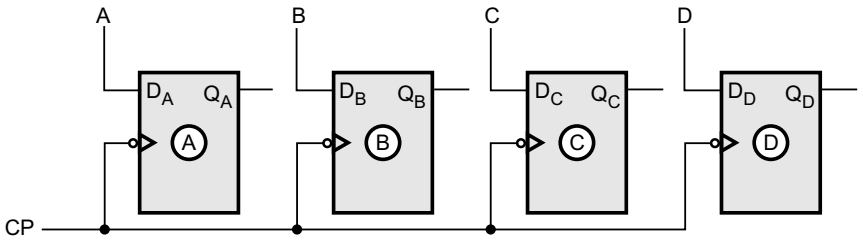


Fig. 1.5.17 Buffer register

Shift Registers

The binary information (data) in a register can be moved from stage to stage within the register or into or out of the register upon application of clock pulses. This type of bit movement or shifting is essential for certain arithmetic and logic operations used in microprocessors. This gives rise to a group of registers called '**shift registers**'. They are very important in applications involving the storage and transfer of data in a digital system.

Fig. 1.5.18 gives the symbolical representation of the different types of data movement in shift register operations.

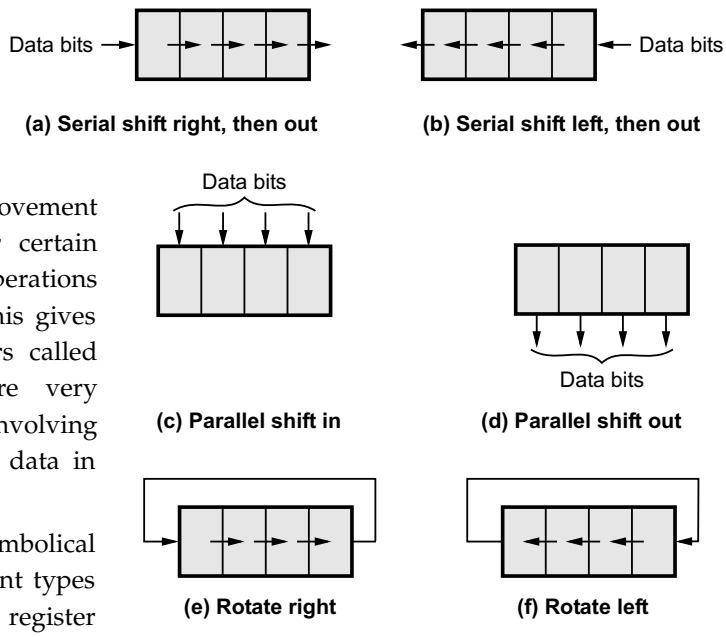
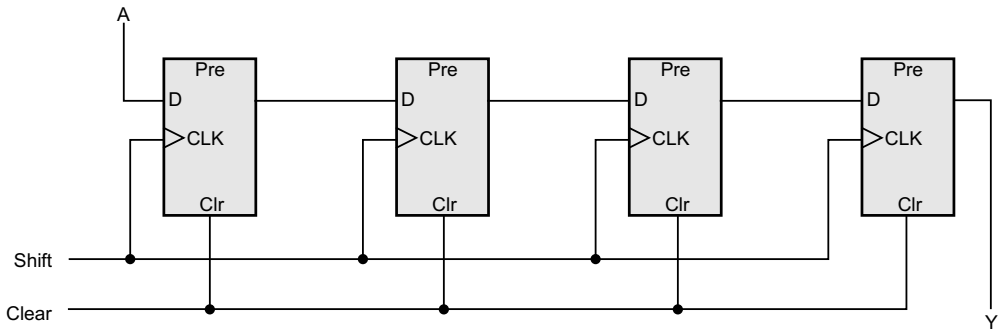
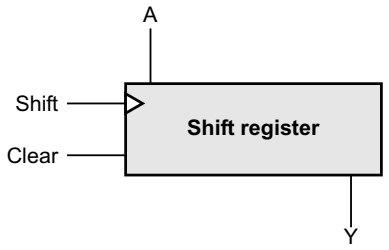


Fig. 1.5.18 Basic data movement in registers



(a) Logic diagram



(b) Symbol

Fig. 1.5.19 4-bit right shift register

The Fig. 1.5.19 shows the register level implementation of right shift register using D flip-flops. A right shift is accomplished by activating the SHIFT enable line connected to the clock input CLK of each flip-flop. In addition to the serial data lines, m input and output lines are often provided to permit parallel data transfers to or from the shift register. Additional control lines are required to select the serial or parallel input modes. The shift register further can be refined to permit both left and right shift operations. The Fig. 1.5.20 shows the shift register with parallel and serial modes along with the right and left shifts operations.

As shown in the Fig. 1.5.20, the D input of each flip-flop has three sources : Output of left adjacent flip-flop, output of right adjacent flip-flop and parallel input. Out of these three sources one source is selected at a time and it is done with the help of decoder. The decoder select lines (SL_1 and SL_0) select the one source out of three as shown in the Table 1.5.5.

SL_1	SL_0	Selected Source
0	0	Parallel input
0	1	Output of right adjacent FF
1	0	Output of left adjacent FF
1	1	–

Table 1.5.5

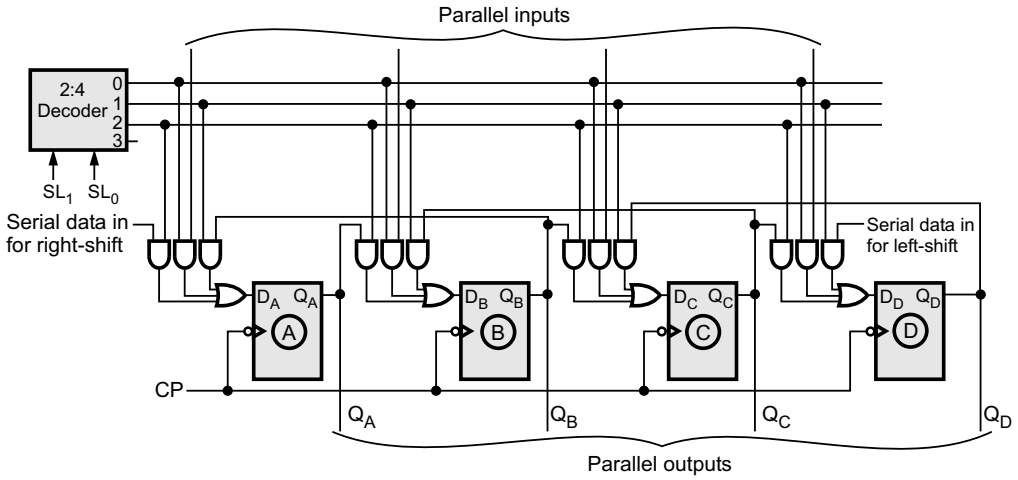


Fig. 1.5.20 4-bit bidirectional shift register with parallel load

When select lines are 00 (i.e. SL_1 = 0 and SL_0 = 0), data from the parallel inputs is loaded into the 4-bit register. When select lines are 01 (i.e. SL_1 = 0 and SL_0 = 1), data within the register is shifted 1-bit left. When select lines are 10 (i.e. SL_1 = 1 and SL_0 = 0), data with in the register is shifted 1-bit right.

Tri-State Register

In the buffer register, there is no control over input as well as output bits. We can control input and output of the register by connecting tri-state devices at the input and output of the register as shown in the Fig. 1.5.21.

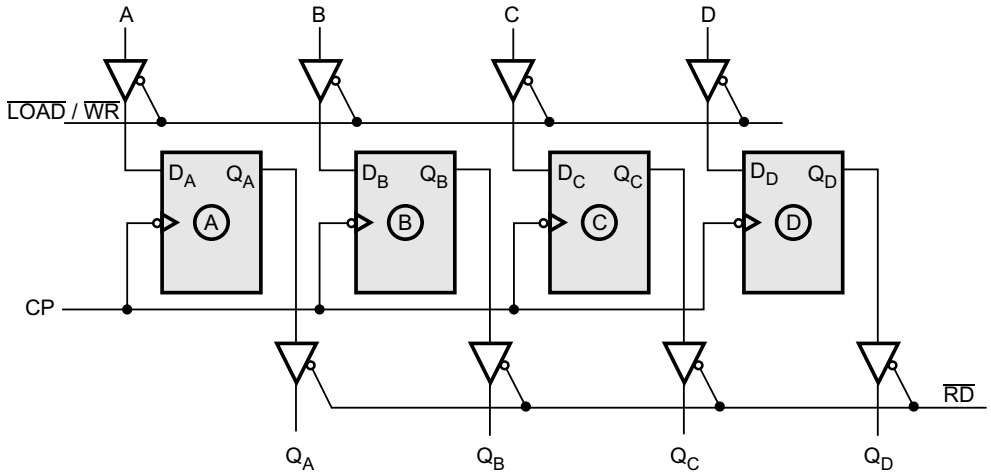


Fig. 1.5.21 Tri-state register

As shown in the Fig. 1.5.21, tri-state switches are used to control the read/write operation. The tri-state switch is a binary switch. It is closed when enabled and it is open, i.e., in the high impedance state when disabled. Here $\overline{RD}$ and $\overline{WR}$ signals are used

as enabled signal. To get the data on the output lines $\overline{RD}$ signal is enabled and to load data into the register $\overline{WR}$ signal is enabled. When $\overline{RD}$ signal is disabled the output lines are in high-impedance state.

Buses

A bus is a group of wires that transmits a binary word. In Fig. 1.5.22 group of wires B_3, B_2, B_1 and B_0 is a bus. The number of wires decide binary word. Thus the bus shown in the Fig. 1.5.22 is four bit bus. The bus is a common transmission path between the tri-state registers. The input for and output from all the registers are connected to the common bus.

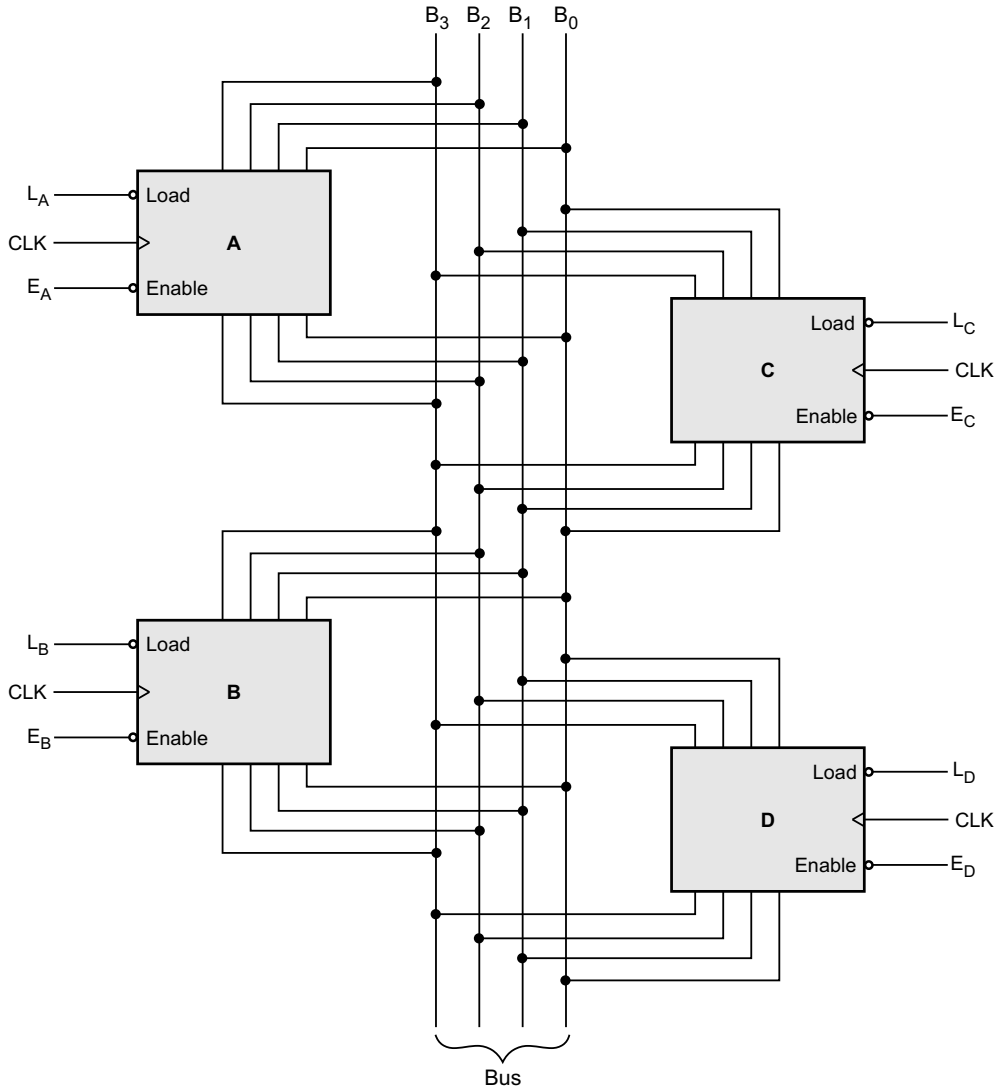


Fig. 1.5.22 Connecting registers to common data bus

In the Fig. 1.5.23 all the control signals are in complemented form; this means that registers have active low inputs.

In this bus organization nothing happens until we apply low input signals. In other words, as long as all LOAD and ENABLE inputs are high, the registers are isolated from the bus. To transfer the word from one register to another, it is necessary to make the appropriate control signals low. For example, to transfer the word from register A to register B, it is necessary to make E_A and L_B inputs low.

The connection between common data bus and registers can be shown in simplified form as shown in the Fig. 1.5.23. Here, the data path is shown by a single line. The number of actual data lines in the data path are indicated by number with a slash on the line. The input and output data lines are made common.

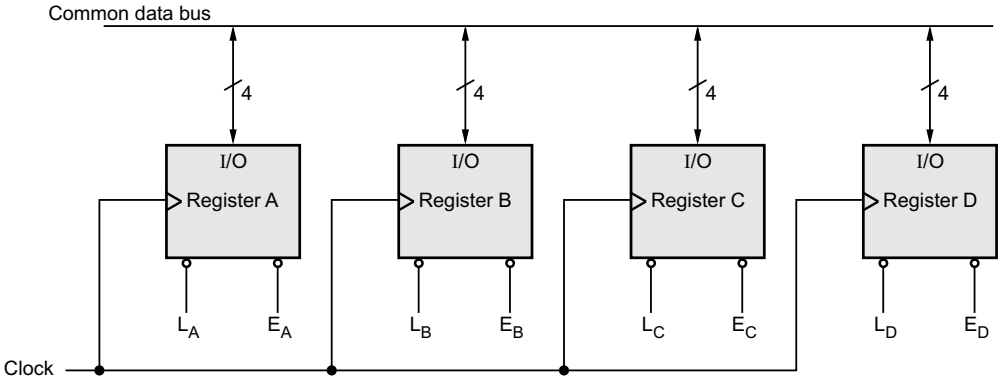


Fig. 1.5.23 Simplified way of showing registers and its connection with common data bus

Counters

A register is used solely for storing and shifting data which is in the form of 1s and/or 0s, entered from an external source. It has no specific sequence of states except in certain very specialized applications. A counter is a register capable of counting the number of clock pulses arriving at its clock input. Count represents the number of clock pulses arrived. A specified sequence of states appears as the counter output. This is the main difference between a register and a counter. A specified sequence of states is different for different types of counters.

There are two types of counters, **synchronous** and **asynchronous**. In synchronous counter, the common clock input is connected to all of the flip-flops and thus they are clocked simultaneously. In asynchronous counter, commonly called, **ripple counters**, the first flip-flop is clocked by the external clock pulse and then each successive flip-flop is clocked by the Q or $\overline{Q}$ output of the previous flip-flop. Therefore in an asynchronous counter, the flip-flops are not clocked simultaneously.

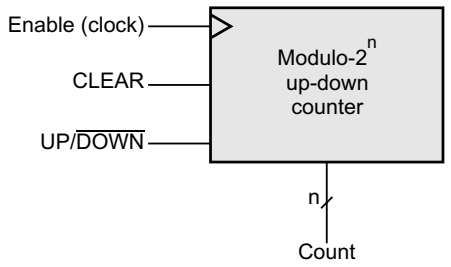


Fig. 1.5.24 A modulo 2^n up-down counter

The Fig. 1.5.24 shows the symbol for a modulo 2^n up-down counter. On receiving positive going edge on the Enable (clock signal) input counter increments its count by 1. As it is an n-bit counter, its counting is modulo-2^n; that is, the counter's modulus $k = 2^n$ and it has 2^n states S_0, S_1, … , S_{2^n-1}. The output of counter is an n-bit binary number. The CLEAR input of the counter when activated resets the counter and UP/$\overline{\text{DOWN}}$ input selects the operation of the counter.

1.5.2 Programmable Logic Devices

There are many applications for digital logic where the market is not great enough to develop a special-purpose MSI or LSI chip. This situation has led to the development of **Programmable Logic Devices** (PLDs) which can be easily configured by the individual user for specialized applications.

Basically, there are three types of PLDs :

- Read Only Memory (ROM)
- Programmable Logic Array (PLA)
- Programmable Array Logic (PAL)

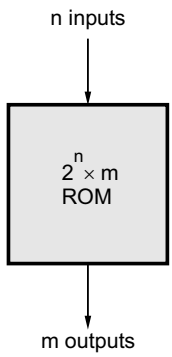


Fig. 1.5.25 Block diagram of ROM

Here, we examine programmable logic devices as a new class of components.

A read only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The Fig. 1.5.25 shows the block diagram of ROM. It consists of n input lines and m output lines. Each bit combination of the input variables is called an **address**. Each bit combination that comes out of the output lines is called a **word**. The number of bits per word is equal to the number of output lines, m. The address specified in binary number denotes one of the minterms of n variables. The number of distinct addresses possible with n input variables is 2^n. An output word can be selected by a unique address and since there are 2^n distinct addresses in a ROM, there are 2^n distinct words in the ROM. The word available on the output lines at any given time depends on the address value applied to the input lines.

Let us consider 64 × 4 ROM. The ROM consists of 64 words of 4 bits each. This means that there are four output lines and particular word from 64 words presently available on the output lines is determined from the six input lines. There are only six inputs in a 64 × 4 ROM because 2^6 = 64 and with six variables, we can specify 64 addresses or minterms. For each address input, there is a unique selected word. Thus, if the input address is 000000, word number 0 is selected and applied to the output lines. If the input address is 111111, word number 63 is selected and applied to the output lines.

The Fig. 1.5.26 shows the internal logic construction of a 64 × 4 ROM. The six input variables are decoded in 64 lines by means of 64 AND gates and 6 inverters. Each output of the decoder represents one of the minterms of a function of six variables. The 64 outputs of the decoder are connected through fuses to each OR gate. Only four of these fuses are shown in the diagram, but actually each OR gate has 64 inputs and each input goes through a fuse that can be blown as desired.

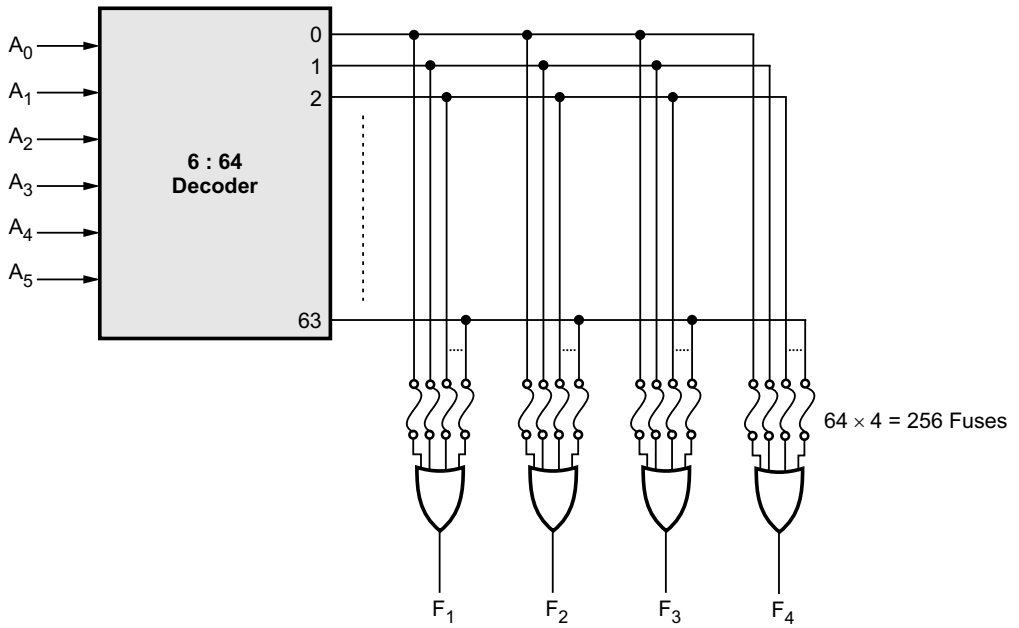


Fig. 1.5.26 Logic construction of 64 × 4 ROM

The ROM is a two level implementation in sum of minterms form. Let us see AND-OR and AND-OR-INVERTER implementation of ROM. Fig. 1.5.27 shows the 4 × 2 ROM with AND-OR and AND-OR-INVERTER implementations.

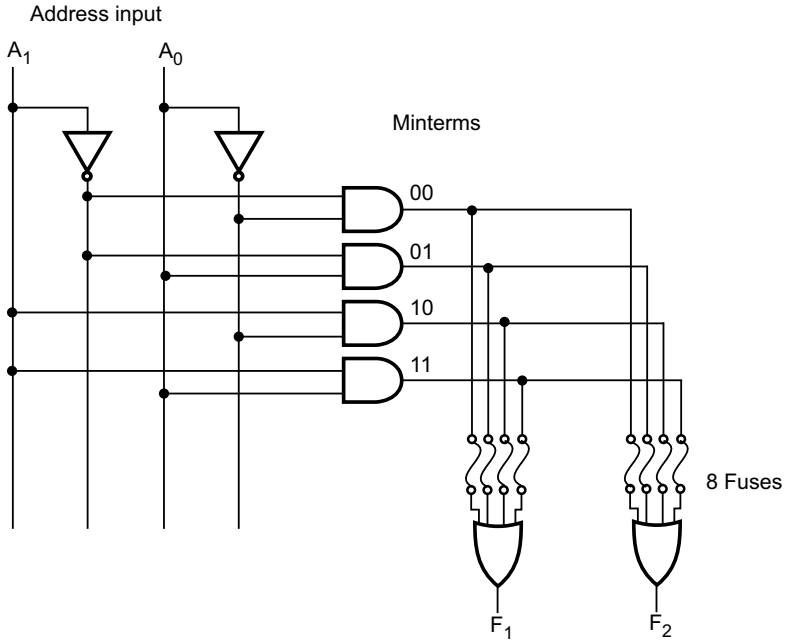


Fig. 1.5.27 (a) 4 × 2 ROM with AND-OR gates

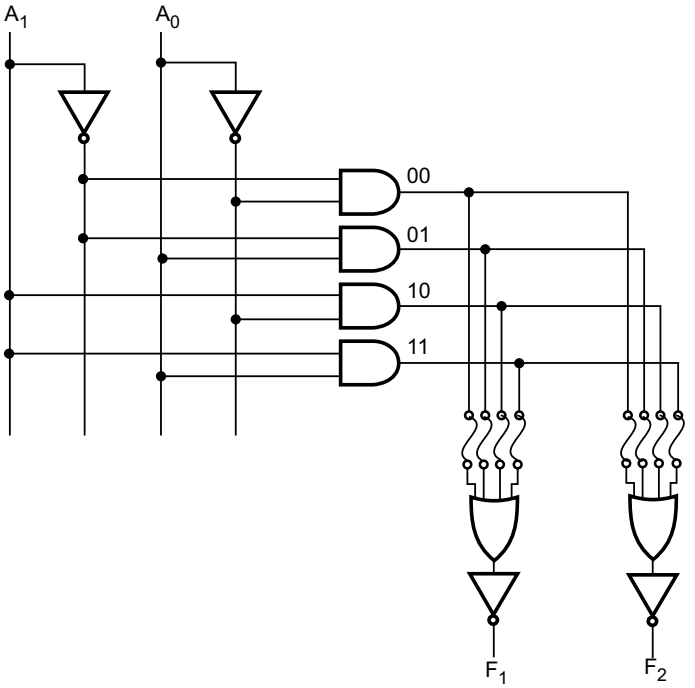


Fig. 1.5.27 (b) 4 × 2 ROM with AND-OR-INVERTER gates

There are four types of ROM : Masked ROM, PROM, EPROM and EEPROM.

PROM (Programmable Read Only Memory)

Programmable Read Only Memory (PROM) allows user to store data/program. PROMs use the fuses with material like nichrome and polycrystalline. The user can blow these fuses by passing around 20 to 50 mA of current for the period 5 to 20 μs. The blowing of fuses according to the truth table is called *programming of ROM*. The user can program PROMs with special PROM programmer. The PROM programmer selectively burns the fuses according to the bit pattern to be stored. This process is also known as *burning of PROM*. The PROMs are one time programmable. Once programmed, the information stored is permanent.

EPROM (Erasable Programmable Read Only Memory)

Erasable programmable ROMs use MOS circuitry. They store 1s and 0s as a packet of charge in a buried layer of the IC chip. EPROMs can be programmed by the user with a special EPROM programmer. The important point for now is that we can erase the stored data in the EPROMs by exposing the chip to ultraviolet light through its quartz window for 15 to 20 minutes.

In EPROMs, it is not possible to erase selective information; when erased, the entire information is lost. The chip can be reprogrammed. This memory is ideally suitable for product development, experimental projects and college laboratories, since this chip can be reused many times.

EEPROM (Electrically Erasable Programmable Read Only Memory)

Electrically erasable programmable ROMs also use MOS circuitry very similar to that of EPROM. Data is stored as charge or no charge on an insulated layer or an insulated floating gate in the device. The insulating layer is made very thin (< 200 $\mathring{A}$). Therefore, a voltage as low as 20 to 25 V can be used to move charges across the thin barrier in either direction for programming or erasing. EEPROM allows selective erasing at the register level rather than erasing all the information since the information can be changed by using electrical signals. The EEPROM memory also has a special chip erase mode by which entire chip can be erased in 10 ms. This time is quite small as compared to time required to erase EPROM and it can be erased and reprogrammed with device right in the circuit. However, EEPROMs are most expensive and the least dense ROMs.

The combinational circuit do not use all the minterms every time. Occasionally, they have don't care conditions. Don't care condition when implemented with a ROM becomes an address input that will never occur. The result is that not all the bit patterns available in the ROM are used, which may be considered a waste of available equipment.

For cases where the number of don't care conditions is excessive, it is more economical to use a second type of LSI component called a **Programmable Logic Array**

(PLA). A PLA is similar to a ROM in concept; however it does not provide full decoding of the variables and does not generates all the minterms as in the ROM. The PLA replaces decoder by group of AND gates, each of which can be programmed to generate a product term of the input variables. In PLA, both AND and OR gates have fuses at the inputs, therefore in PLA both AND and OR gates are programmable. Fig. 1.5.28 shows the block diagram of PLA. It consists of n inputs, m outputs, k product terms, and m sum terms. The product terms constitute a group of k AND gates and the sum terms constitute a group of m OR gates. Fuses are inserted between all n inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gates and the inputs of the OR gates. The third set of fuses in the output inverters allows the output function to be generated either in the AND-OR form or in the AND-OR-INVERT form. When inverter is bypassed by link we get AND-OR implementation. To get AND-OR-INVERTER implementation inverter link has to be disconnected.

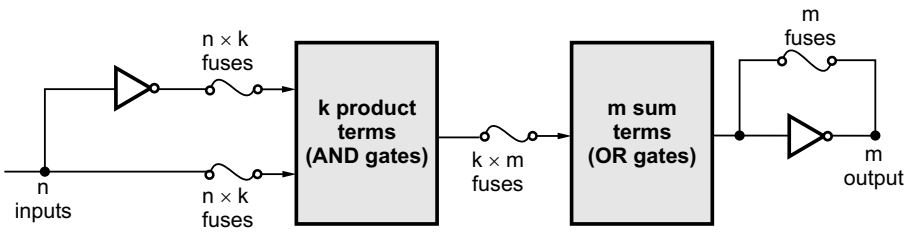


Fig. 1.5.28 Block diagram of PLA

Fig. 1.5.29 shows the internal construction of PLA having 3 inputs, 3 product terms and two outputs. The size of the PLA is specified by the number of inputs, the number of product terms and the number of outputs (the number of sum terms is equal to the number of outputs.)

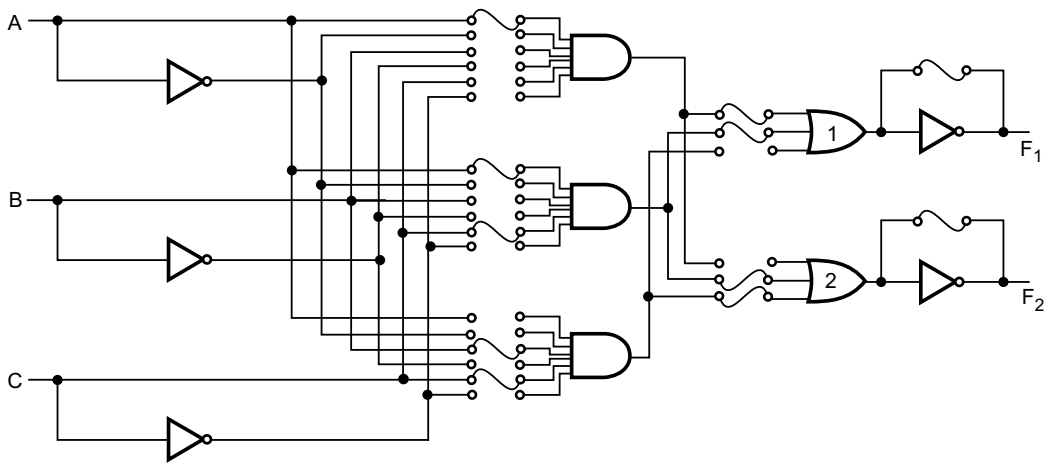


Fig. 1.5.29 Internal construction of PLA

Like ROM, PLA can be **mask-programmable** or **field-programmable**. With a mask-programmable PLA, the user must submit a PLA program table to the manufacturer. This table is used by the vendor to produce a user-made PLA that has the required internal paths between inputs and outputs. A second type of PLA available is called a **field-programmable logic array** or FPLA. The FPLA can be programmed by the user by means of certain recommended procedures. FPLAs can be programmed with commercially available programmer units.

Programmable logic devices have many gates interconnected through many of electronic fuses. It is sometimes convenient to draw the internal logic of such devices in a compact form referred to as **array logic**. Fig. 1.5.30 shows the conventional and array logic symbols for a multiple-input AND gate.



Fig. 1.5.30

The array logic symbol shown in the Fig. 1.5.30 (b) uses a single horizontal line connected to the gate input and multiple vertical lines to indicate the individual inputs. Each intersection between horizontal line and vertical line indicates the fuse connection.

We have seen that PLA is a device with a programmable AND array and programmable OR array. However, PAL programmable array logic is a programmable logic device with a fixed OR array and a programmable AND array. Because only AND gates are programmable, the PAL is easier to program, but is not as flexible as the PLA. Fig. 1.5.31 shows the array logic of a typical PAL. It has four inputs and four outputs. Each input has buffer and an inverter gate. It is important to note that two gates are shown with one composite graphic symbol with normal and complement outputs. There are four sections. Each section has three programmable AND gates and one fixed OR gate. As shown in the Fig. 1.5.31, each AND gate has 10 fused programmable inputs. The output of section 1 is connected to a buffer-inverter gate and then fedback into the inputs of the AND gates through fuses.

The commercial PAL devices has more gates than the one shown in Fig. 1.5.31. A typical PAL integrated circuit may have eight inputs, eight outputs and eight sections, each consisting of an eight wide AND-OR array.

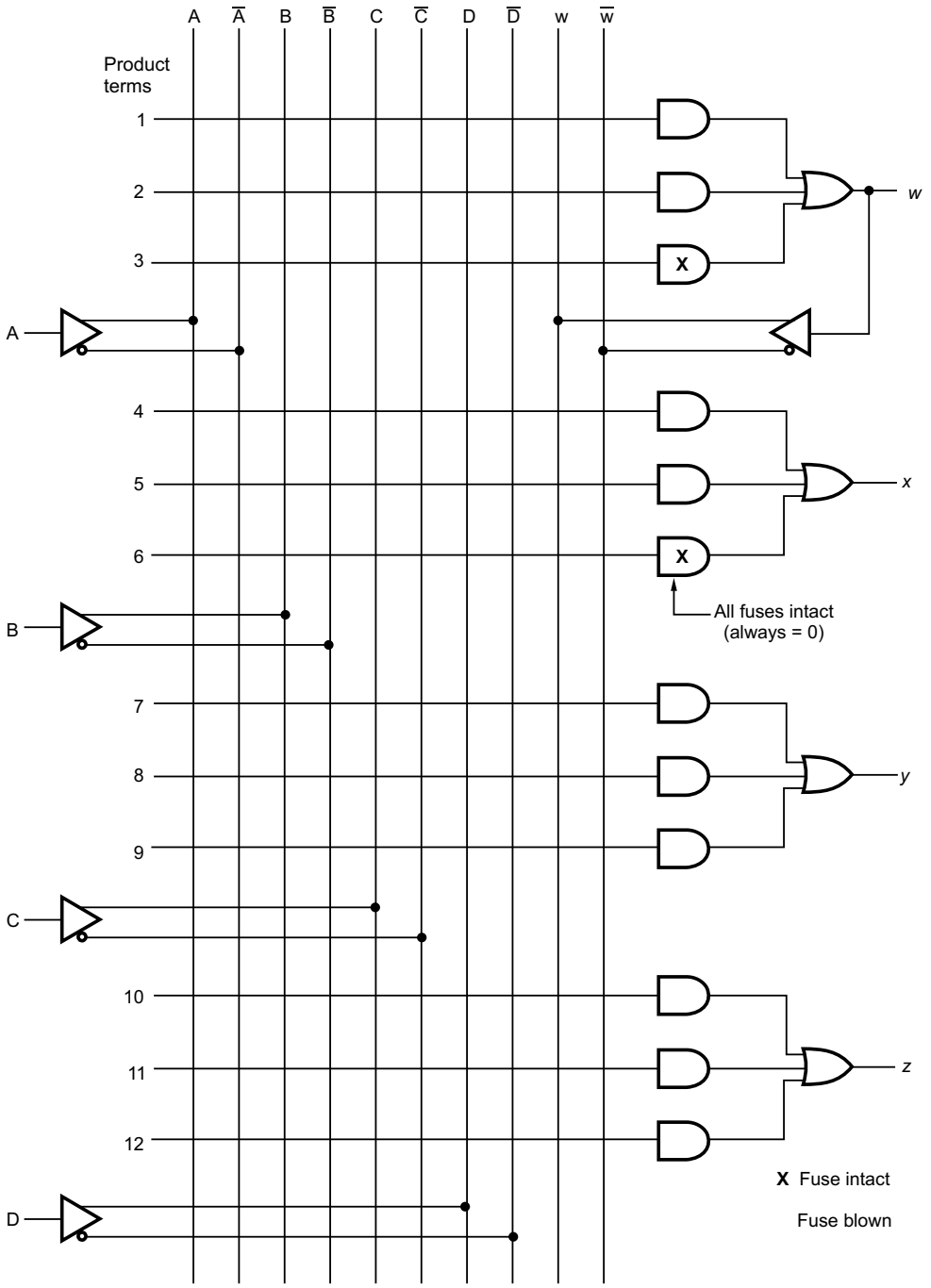

A
$\overline{A}$
B
$\overline{B}$
C
$\overline{C}$
D
$\overline{D}$
w
$\overline{w}$
Product terms
1
2
3
4
5
6
7
8
9
10
11
12
A
B
C
D
w
x
y
z
X
All fuses intact (always = 0)
X Fuse intact
Fuse blown

Fig. 1.5.31

1.5.3 Field Programmable Gate Arrays

In mid-1980s, an important class of PLDs was introduced, called field-programmable gate array. The Fig. 1.5.32 shows the general structure of FPGA chip. It consists of a large number of programmable logic blocks surrounded by programmable I/O block. The programmable logic blocks of FPGA are smaller and less capable than a PLD, but an FPGA chip contains a lot more logic blocks to make it more capable. As shown in the Fig. 1.5.32, the logic blocks are distributed across the entire chip. These logic blocks can be interconnected with programmable inter connections.

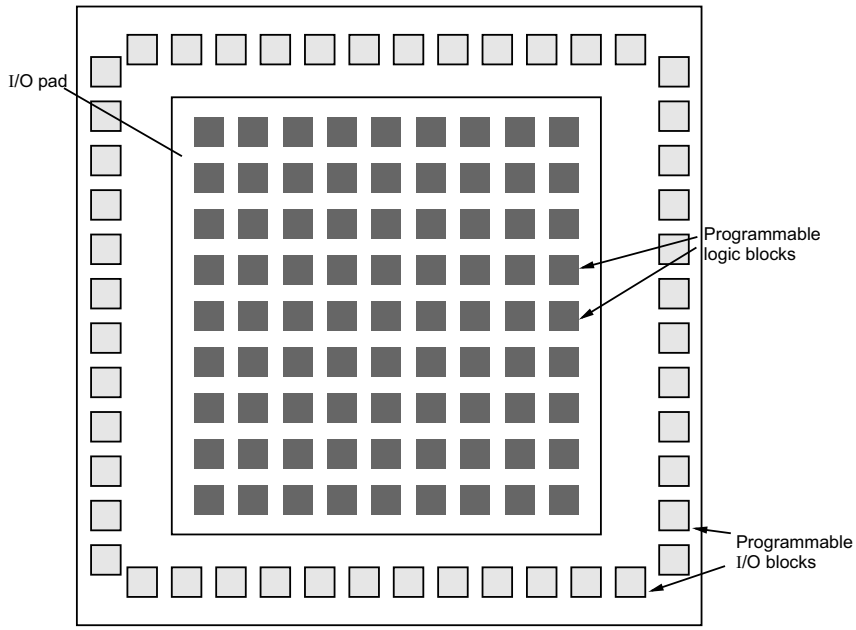


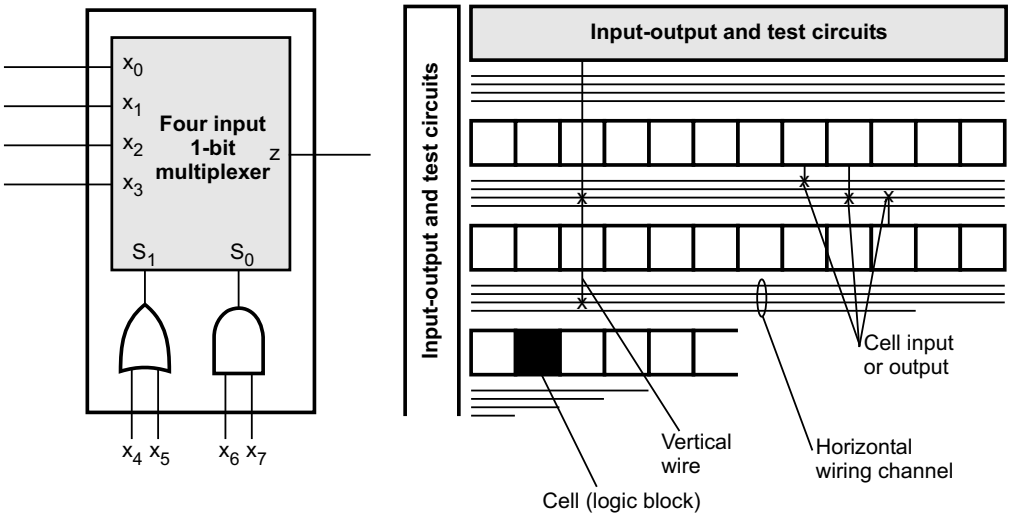
Fig. 1.5.32 General FPGA chip architecture

As compared to standard gate arrays, the field programmable gate arrays are larger devices. The basic cell structure for FPGA is some what complicated than the basic cell structure of standard gate array. The FPLA use read/write memory cell to control the state of each connection.

The word **field** in the name refers to the ability of the gate arrays to be programmed for a specific function by the user instead of by the manufacturer of the device. The word **array** is used to indicate a series of columns and rows of gates that can be programmed by the end user.

Two types of logic cells found in FPGAs are those based on multiplexers and those based on PROM table-lookup memories. Fig. 1.5.33 shows a cell type employed by Actel Corp.'s ACT series of multiplexer-based FPGAs. This cell is a four-input, 1-bit multiplexer with an AND and OR gate added. An ACT FPGA contains a large array of

such cells organized in rows separated by horizontal wiring channels as shown in Fig. 1.5.33 (b). Vertical wire segments are attached to each cell's I/O terminal. These wires enable connections to be established between the cells and the wiring channels by means of one-time programmable antifuses positioned where the horizontal and vertical wires cross.



(a) Basic cell (b) Chip architecture

Fig. 1.5.33 Multiplexer based FPGA

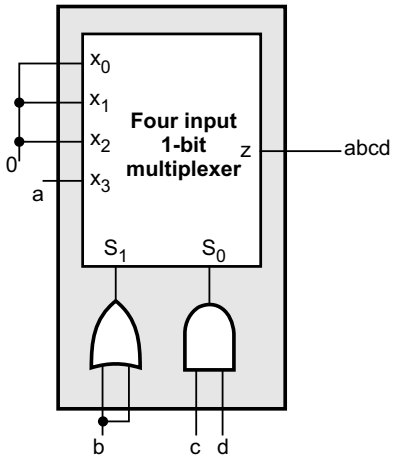


Fig. 1.5.34 (a) AND gate

We know that multiplexer can be used as a function generator; it can be used to implement any Boolean function. Therefore, the cell in the multiplexer based FPGA is also capable of implementing various useful Boolean functions. Fig. 1.5.34 shows the implementation of complete set of logic gates using this cell.

$$Z = x_0 \bar{s}_1 \bar{s}_0 + x_1 \bar{s}_1 s_0 + x_2 s_1 \bar{s}_0 + x_3 s_1 s_0$$

$$= a\, s_1 s_0 \qquad \because x_0 = x_1 = x_2 = 0 \text{ and } x_3 = a$$

$$= abcd \qquad \because s_1 = b \text{ and } s_0 = cd$$

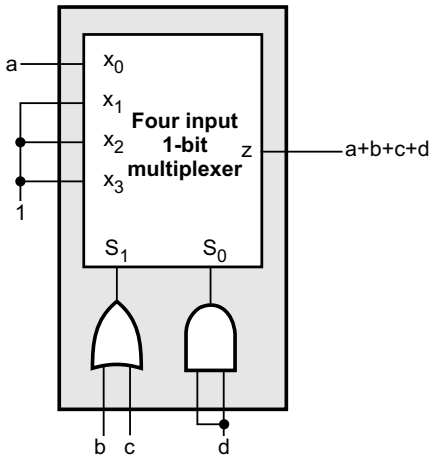


Fig. 1.5.34 (b) OR gate

$$Z = x_0 \bar{s}_1 \bar{s}_0 + x_1 \bar{s}_1 s_0 + x_2 s_1 \bar{s}_0 + x_3 s_1 s_0$$

$$= a \bar{s}_1 \bar{s}_0 + \bar{s}_1 s_0 + s_1 \bar{s}_0 + s_1 s_0$$

$\therefore x_1 = x_2 = x_3 = 1$ and $x_0 = a$

$$= a \bar{s}_1 \bar{s}_0 + s_0 + s_1 \bar{s}_0 \qquad \because A + \bar{A} = 1$$

$$= a \bar{s}_1 \bar{s}_0 + s_0 + s_1 \qquad \because A + \bar{A} B = A + B$$

$$= a\overline{(b+c)}\,\bar{d} + d + (b+c)$$

$\because s_1 = b + c$ and $s_0 = d$

$$= a\overline{(b+c)} + d + (b+c) \qquad \because A + \bar{A} B = A + B$$

$$= a + d + b + c \qquad \because A + \bar{A} B = A + B$$

$$= a + b + c + d$$

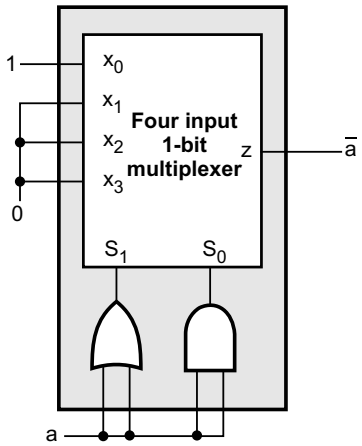


Fig. 1.5.34 (c) NOT gate

$$Z = x_0 \bar{s}_1 \bar{s}_0 + x_1 \bar{s}_1 s_0 + x_2 s_1 \bar{s}_0 + x_3 s_1 s_0$$

$$= \bar{a}\,\bar{a} = \bar{a}$$

$\because x_0 = 1, x_1 = x_2 = x_3 = 0, s_1 = a$ and $s_0 = a$

1.5.4 Register Level Design

At register level design, a set of registers are linked by combinational data transfer and data-processing circuits. A block diagram defines its structure and the set of operations it performs on data words can define its behaviour. Each operation can be defined in the form :

cond : $Z = f(A_1, A_2, A_3, \ldots, A_k)$

where f is a function to be performed or an instruction to be executed in one clock cycle, and A_1, A_2, A_3, ... , A_k and Z denote data words or the registers that store data. The prefix 'cond' denotes a control condition that must be satisfied (cond = 1) for the

indicated operation to take place. Therefore, when 'cond' = 1 the function f is computed on A_1, A_2, A_3, ... and A_k and result is stored in the Z data word.

1.5.4.1 Data and Control

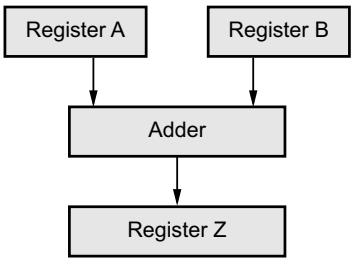


Fig. 1.5.35 (a) Simple register level system

The Fig. 1.5.35 (a) shows simplest register level system. It performs operation Z = A + B. The Fig. 1.5.35 (b) shows a more complicated system that can perform several different operations. Such a multifunction system can perform only one operation at a time. The operation to be perform is decided by the control signals. Therefore, the multifunction system is partitioned into a data-processing part called a **datapath** and a controlling part called a **control unit**. The control unit is responsible for selecting and controlling the actions of the datapath.

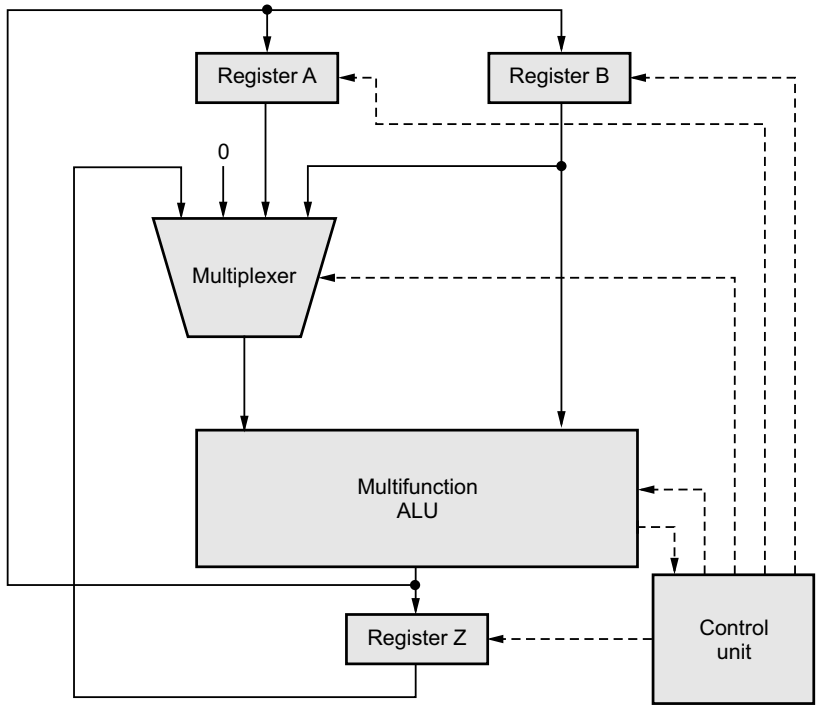


Fig. 1.5.35 (b) Multifunction register level system

As shown in Fig. 1.5.35 (b), control unit (CU) selects the operation for the ALU to perform in each clock cycle. It also determines the input operands to apply to the ALU and the destination of its results.

The large extension to this multifunction unit is the computer's CPU. The computer's CPU is responsible for the interpretation of instructions and generation of required control signals for the execution of the instructions. This unit of computer is called **I-unit** and the datapath unit of computer is called **E-unit**.

1.5.4.2 A Description Language

The HDL can be used to provide both behavioural and structural description at the register level. For example,

if cond = 1 **then** Z = f (A_1, A_2, A_3, ... , A_k) ;

where f can be any function. For example, Z = A + B or Z = A – B. Here, + represents the adder and – represents the subtractor. The input connections in both the cases from registers A and B are inferred from the fact that A and B are the arguments of + and –, while the output connection from the adder/subtractor is inferred from Z = .

Let us see the formal language description of an 8-bit binary multiplier. We know that, the multiplication can be perform in two ways : 1. Repetitive addition 2. Shift and add. Here, our intention is to study the language descriptions, hence we prefer simple method of multiplication, i.e. multiplication by repetitive addition.

```
Multiplication   (in : INBUS; out : OUTBUS);
                 register A [0:7], B [0:7], Z [0:7],
                 C [0:7]; bus INBUS [0:7],
                 OUTBUS [0:7];
BEGIN :          Z = 0, C = 0, A = INBUS ;
                 B = INBUS ;
REPEAT :         Z = Z + A ;
                 if CY ≠ 1 then go to NEXT
                 C = C + 1;
NEXT :           B = B – 1 ;
                 if B ≠ 0 then go to REPEAT
                 OUTBUS = Z ;
                 OUTBUS = C ;
end multiplication ;
```

In the above program, two 8-bit buses INBUS and OUTBUS form multipliers input and output ports, respectively. The program initializes these buses with statement : bus INBUS [0 : 7], OUTBUS [0 : 7] and initialize 8-bit registers A, B, Z and C with statement : register A [0 : 7], B [0 : 7], Z [0 : 7] and C [0 : 7]. The registers Z and C are initialized to store the lower byte and higher byte of multiplication, respectively. Initially, the result (Z and C) is made 0 and register A and B are loaded with multiplicand and multiplier from the INBUS, respectively. The multiplicand is added repeated for multiplier times and result is stored in the Z and C registers. The carry after addition of lower byte is used to increment the value in the higher byte register, i.e. C register. The final result is then transferred 8 bits at a time to OUTBUS.

1.5.4.3 Design Techniques

The general approach to the design problem for register level system is as follows :

1. Define the desired behaviour of the system by a set of sequences of register-transfer operations, such that each operation can be implemented directly using the available design components. This gives the desired algorithm.
2. Analyse the algorithm, to determine the types of components and the number of each type required for the datapath.
3. Construct a block diagram for datapath using the components identified in step 2. Make the connections between the components so that all data paths implied by algorithm are present and the given performance-cost constraints are met.
4. Analyse algorithm and datapath to identify the control signals needed. Introduce the logic or control points necessary to apply these signals into datapath.
5. Design a control unit for datapath that meets all the requirements of algorithm.
6. Check whether the final design operates correctly and meets all performance-cost goals.

A design of algorithm in step 1 is a creative design process. It is similar to writing computer program and depends heavily on the skill and experience of the designer. The second step is to identify the data processing components. It is a straight forward process. However, it becomes complicated when the possibility of sharing components exists. For example, the perform operation

Cond : A = A + B, C = C + D ;

requires two adders, because the operation has to perform in parallel. However, if we use single adder and perform operations serially we can lower the cost by sharing a single adder. Thus :

Cond (t_0) : A = A + B

Cond (t_0 + 1) : C = C + D

The step 3 requires defining an interconnection structure that links the component needed by the various parts of algorithm. Identifying the control signals and design of control unit in step 4 and step 5, respectively, is a relatively independent process. The step 6, design verification plays an important role in the development process. The simulation via CAD tool can be used to identify and correct functional errors before new design is committed to hardware.

Review Questions

1. *List various register level component.*
2. *Draw and explain the generic block representation of a register level component.*
3. *Write short notes on*
   a) Multiplexers b) Decoder c) Encoder
   d) Demultiplexers e) Arithmetic elements f) Registers
   g) Tristate register h) Buses i) Counters
   j) Programmable logic designs k) FPLA
4. *Explain the design process at the register level.*
5. *Write short notes on HDL.*
6. *Explain the register-level design of a 4-bit magnitude comparator.* **Dec.-03, Marks 8**
7. *Describe the organization of a processor with the general register organization.* **Dec.-03, Marks 8**
8. *What is a priority encoder ? Design a 16-bit priority encoder using two copies of an 8-bit priority encoder.* **May-11, Marks 6**
9. *Design a 4-bit bidirectional shift register with parallel load and explain.* **May-11, Marks 10**
10. *Draw the block diagram of a 4 bit register level magnitude comparator and explain.* **Dec.-11, Marks 8**

1.6 Processor Level

Dec.-03, 11, May-11

The processor level which is also called **system level** is the highest in the hierarchy of computer design. The storage and processing of information are the major objectives of this level. Processing involves execution of programs and processing of data files. The components required for performing these functions are complex. Usually sequential circuits are used which are based on VLSI technology. A slight design theory is necessary at this level of abstraction.

1.6.1 Processor Level Components

The different types of components which are generally used at this level can be divided mainly into four groups as

- Processors
- Memories
- I/O devices
- Interconnection networks

In this section we will see the brief summary of the characteristics of all these components.

1.6.1.1 Central Processing Unit

The primary function of a central processing unit is to execute sequences of instructions stored in a memory, which is external to the central processing unit. When the functions of the processor are restricted, those processors become more specialized processor such as I/O processor. Most of the times CPUs are microprocessors whose physical implementation is a single VLSI chip.

Fig. 1.6.1 shows a typical CPU structure and its connection to memory. The CPU contains different units such as control unit, arithmetic logic unit, register unit, decoding unit which are necessary for the execution of instructions. The sequence of operations involved in processing an instruction constitutes an instruction cycle. This can be subdivided into three major phases : fetch cycle, decode cycle and execute cycle. The address of the next instruction which is to be fetched from memory is in the Program Counter (PC). During fetch phase CPU loads this address in Address Register (AR). This is the register which gives address to the memory. Once the address is available on the address bus, the read command from control unit copies the contents of addressed memory location to the instruction register (IR). During decode phase, the instruction in the IR is decoded by instruction decoder. In the next, i.e. execute phase CPU has to perform a particular set of micro-operation depending on the instruction.

All these operations are synchronized with the help of clock signal. The frequency of this signal is nothing but the operating frequency of CPU. Thus the CPU is a synchronous sequential circuit and its clock period is the computer's basic unit of time.

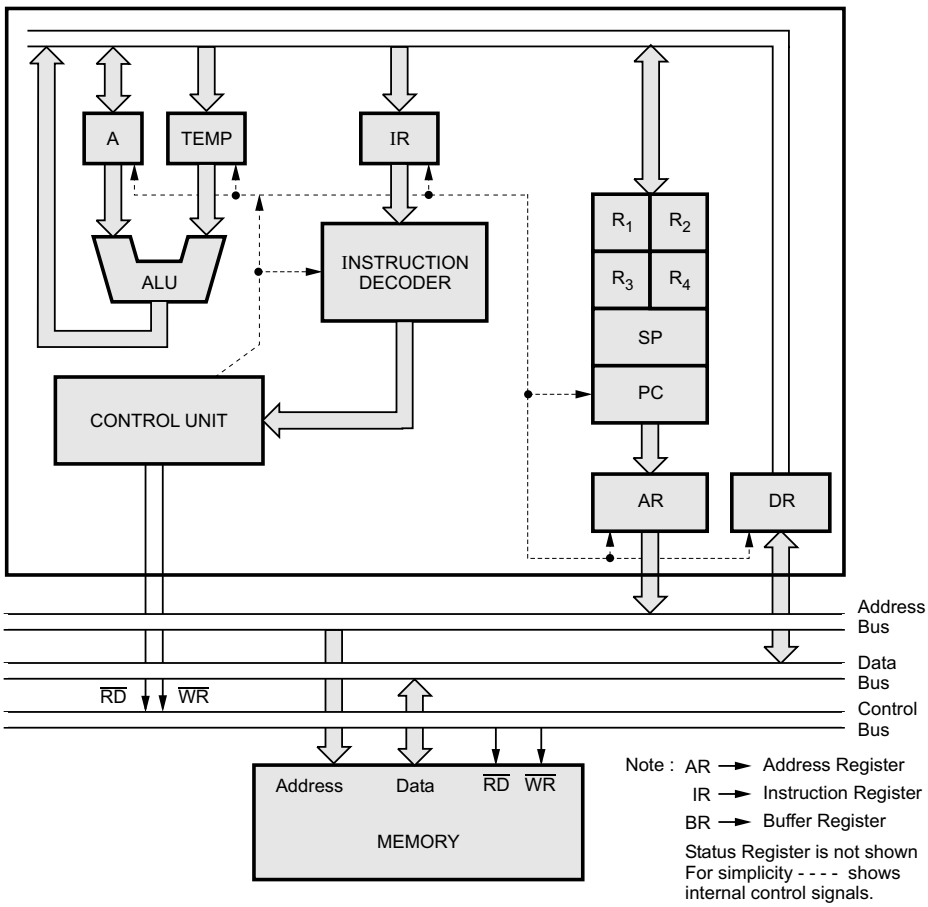


Fig. 1.6.1 Typical CPU structure

1.6.1.2 Memories

For the storage of programs and data required by the processors, external memories are necessary. Ideally, computer memory should be fast, large and inexpensive. Unfortunately, it is impossible to meet all the three of these requirements simultaneously. Increased speed and size are achieved at increased cost. Very fast memory system can be achieved if SRAM chips are used. These chips are expensive and for the cost reason it is impracticable to build a large main memory using SRAM chips. The only alternative is to use DRAM chips for large main memories.

Processor fetches the code and data from the main memory to execute the program. The DRAMs which form the main memory are slower devices. So it is necessary to insert wait states in memory read/write cycles. This reduces the speed of execution. The solution for this problem is come out with the fact that most of the computer programs work with only small sections of code and data at a particular time. In the memory system small section of SRAM is added along with main memory, referred to as *cache memory*. The program which is to be executed is loaded in the main memory, but the part of program (code) and data that work at a particular time is usually accessed from the cache memory. This is accomplished by loading the active part of code and data from main memory to cache memory. The cache controller looks after this swapping between main memory and cache memory with the help of DMA controller. The cache memory just discussed is called *secondary cache.* Recent processors have the built-in cache memory called *primary cache.*

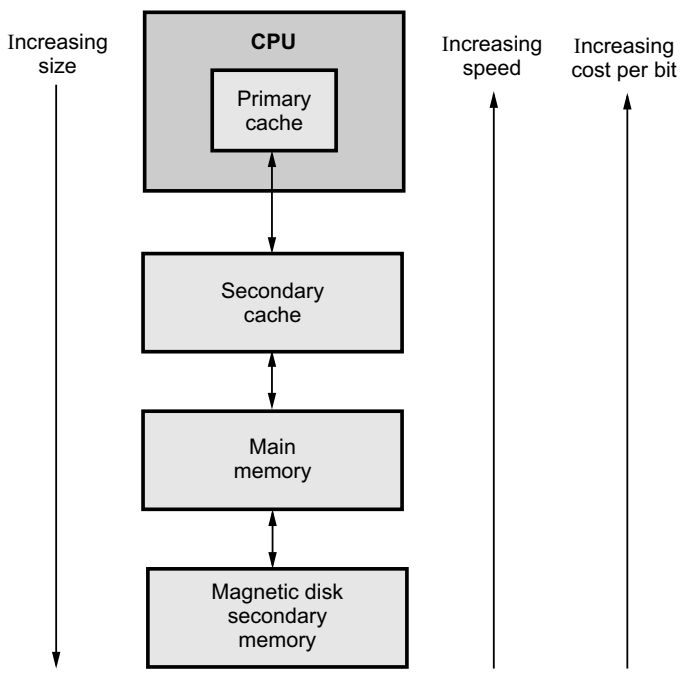


Fig. 1.6.2

DRAMs along with cache allow main memories in the range of tens of megabytes to be implemented at a reasonable cost, the size and better speed performance. But the size of memory is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks and magnetic

tapes to implement large memory spaces. Very large disks are available at a reasonable price, sacrificing the speed.

From the above discussion, we can realize that to make efficient computer system it is not possible to rely on a single memory component, but to employ a *memory hierarchy.* Using memory hierarchy all of different types of memory units are employed to give efficient computer system. A typical memory hierarchy is illustrated in Fig. 1.6.2.

In summary, we can say that a huge amount of cost-effective storage can be provided by magnetic disks. A large, yet affordable, main memory can be built with DRAM technology along with the cache memory to achieve better speed performance.

1.6.1.3 IO Devices

A computer communicates with outside world by means of input-output (IO) system. The main function of I/O system is to transfer information between CPU or memory and the outside world.

The important point to be noted here is, I/O devices (peripherals) cannot be connected directly to the system bus. The reasons are discussed here.

- A variety of peripherals with different methods of operation are available. So it would be impractical to incorporate the necessary logic within the CPU to control a range of devices.
- The data transfer rate of peripherals is often much slower than that of the memory or CPU. So it is impractical to use the high speed system bus to communicate directly with the peripherals.
- Generally, the peripherals used in a computer system have different data formats and word lengths than that of CPU used in it.

So to overcome all these difficulties, it is necessary to use a module in between system bus and peripherals, called as I/O module or I/O system.

This I/O system has two major functions,

- Interface to the CPU and memory via the system bus,
- Interface to one or more I/O devices by tailored data links

The table gives list of representative IO devices.

IO device	Type	Medium to/from which IO device transforms digital electrical signals
Analog-digital converter	I	Analog (continuous) electrical signals
CD-ROM drive	I	Characters (and coded images) on optical disk
Document scanner/reader	I	Images on paper

Dot-matrix display panel	O	Images on screen
Keyboard/keypad	I	Characters on keyboard
Laser printer	O	Images on paper
Loudspeaker	O	Spoken words and sounds
Magnetic-disk drive	I/O	Characters (and coded images) on magnetic disk
Magnetic-tape drive	I/O	Characters (and coded images) on magnetic tape
Microphone	I	Spoken words and sounds
Mouse/touchpad	I	Spatial position on pad

Table 1.6.1

1.6.1.4 Interconnection Networks

The processor level components, CPU, memories, IO devices communicate via system bus (address bus, data bus and control bus). In a computer system, when many components are used, communication between these components may be controlled by a subsystem called an **interconnection network**. Switching network, communications controller, bus controller are the examples of such subsystem. Under the control of interconnection network, dynamic communication paths among the components via the buses can be established. The communication paths are shared by the components to reduce cost. At any time, communication and hence use of shared bus is possible between any two components. When more than two components request use of the bus, it results in bus contention. The function of the interconnecting network is to resolve such contention. For performing this function, interconnecting network selects one of the requesting devices on some priority basis and connects it to the bus. The remaining requesting devices are kept in a queue.

Some evolutionary steps in the I/O function are summarized here.

1. In a simple microprocessor-controlled devices, a peripheral device is directly controlled by CPU.
2. A controller is then added to CPU to control peripheral devices with programming facility.
3. Now interrupts are employed in the configuration mentioned in step 2. This saves the CPU time which was required for a polling of I/O device.
4. DMA controller is introduced to give direct access to memory for I/O module.
5. The I/O module is then enhanced to become a processor with a specialized instruction set tailored for I/O. The I/O processor is capable of executing an I/O

program in memory with directions given by CPU. It can execute I/O program without intervention of CPU.

6. The I/O module is further enhanced to have its own local memory. This makes it possible to control a large set of I/O devices with minimal CPU involvement.

In step 5 and step 6 we have seen that the I/O module is capable of executing programs. Such I/O module is commonly known as **I/O channel**.

Generally the communication between processor-level components is asynchronous since they cannot access some unit or bus simultaneously and hence components cannot be synchronized directly by a common clock signal. The following different causes can be stated regarding this synchronization problem.

- The speed of operation of different components vary over a wide range. E.g. CPUs are faster than main memories and main memories are faster than I/O devices.
- The different components work more independently. E.g. Execution of different programs by CPUs and IOPs.
- It is practically difficult to allow synchronous transmission of information between components due to large physical distance between them.

1.6.2 Processor-Level Design

While designing any system, it is very much difficult to give a precise description of the desired system behaviour. Because of this reason, the processor level design job is critical as compare to register level design. Generally to design at this level, a prototype design of known performance is taken. Then according to the necessity new technologies are added and new performance requirements are achieved.

Performance characteristics : Year by year, the cost of computer systems continues to drop dramatically, while the performance and capacity of those systems continue to rise equally dramatically. In this section we introduce some basic aspects of computer system performance characteristics. The total time needed to execute application programs is the most important measure of computer system performance. In other words we can say that the speed of the computer system is an important characteristics to define the performance of the computer system. The speed of the computer system depends on various factors. Let us discuss those factors.

Hardware : The speed of the processor used in the computer systems basically decides the speed of the computer system. For example, system having processor Pentium IV runs faster than system having Pentium I. However, system speed is not only depends on the processor speed, but it is also affected by the supported hardware.

Each processor has its own address bus width and data bus width, internal registers, on chip memory and the instruction set. Higher data bus width allows the transfer of

data with more number of bit at a time. For example, data bus width of 64-bit allows 64 bits transfer of data at a time and data bus width of 32-bit allows 32-bits transfer of data at a time. Higher address bus width gives higher addressing capacity. More number of internal registers allow to store partial results and avoid unnecessary memory accesses resulting faster operation. Similarly, on-chip memory allows to store currently executing program module, required data and partial results in CPU itself which can be accessed quickly resulting faster operation. The system speed is also depends on the speed of the secondary memory, the speed of the I/O parts and the speed of data transfer between them.

Programming language : Now a days, programs are usually written in a high-level languages. These languages require separate compiler to translate programs into machine level language. Therefore, the computer system is affected by the performance of the compiler and hence the language used for the program.

Pipelining : The processor executes the instruction in steps or phases such as fetch, decode and execute. By overlapping these phases of successive instructions we can achieve a substantial improvement in the performance of the computer system. This technique is known as pipelining.

Parallelism : It is possible to perform transfers to and from secondary memory like storage disk or tapes in parallel with program execution in the processor or with activity in other I/O devices. Such technique is known as parallelism. The most of computer system use this parallelism to the improve the system performance.

Types of memory and IO devices : The performance of the computer depends on the type of memory and IO devices supported by it.

Compatibility with other types of computer cost : The performance of a computer system is also decided by the total cost of the system.

All these performance specifications are considered while designing a new computer system. Eventhough the new computer design is closely based on a known design, accurate performance prediction of the new system may not be possible. For accurate performance, the understanding of the relation between the structure of a computer and its performance is very important. Using some mathematical analysis, a little amount of useful performance evaluation can be done. For performance evaluation, experiments during the design process are to be performed. For this purpose computer simulation can be used or performance of the copy of the machine under working conditions can be measured.

1.6.2.1 Prototype Structures

The processor-level design using prototype-structures involve following steps in the design process.

1. First select a prototype design as per the system requirements and adapt it to satisfy the given performance constraints.

2. Determine the performance of proposed system.
3. If the performance is unsatisfactory, design is to be modified. Repeat step 1.
4. The above steps are to be continued until the acceptable design is obtained and the desired performance constraints are achieved.

These steps are widely followed for designing a computer system. While designing new systems, the precautions are always taken to remain compatible with existing hardware and software standards. The reason is that when these standards are to be changed, computer owners have to spend money to retrain users and programmers. Also the well tested software is to be replaced by the modified software. So in the new design of the computer system the drastic changes in the previous design are generally avoided. Because of all these reasons, there is slow evolution of computer architecture.

Fig. 1.6.3 shows the structure of first-generation computers. This is the basic computer structure.

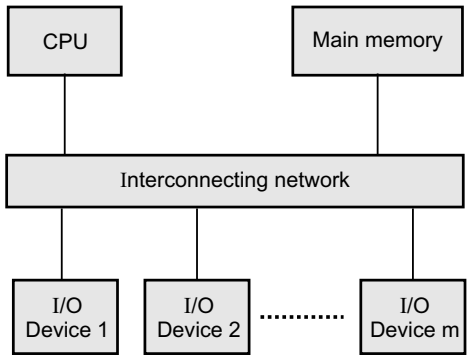


Fig. 1.6.3 Basic computer structure

The second and subsequent generations of computer involve special-purpose IO processors and cache memory in addition to basic components used within the basic system. This advanced structure is shown in Fig. 1.6.4.

The more advanced structure involves more than one CPU, i.e. a multiprocessor system. Fig. 1.6.5 gives the computer structure with two CPUs, main memory banks.

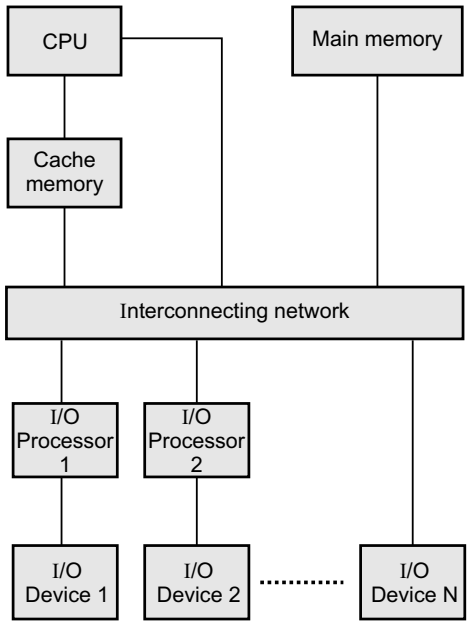


Fig. 1.6.4 Computer structure with IO processors and cache memory

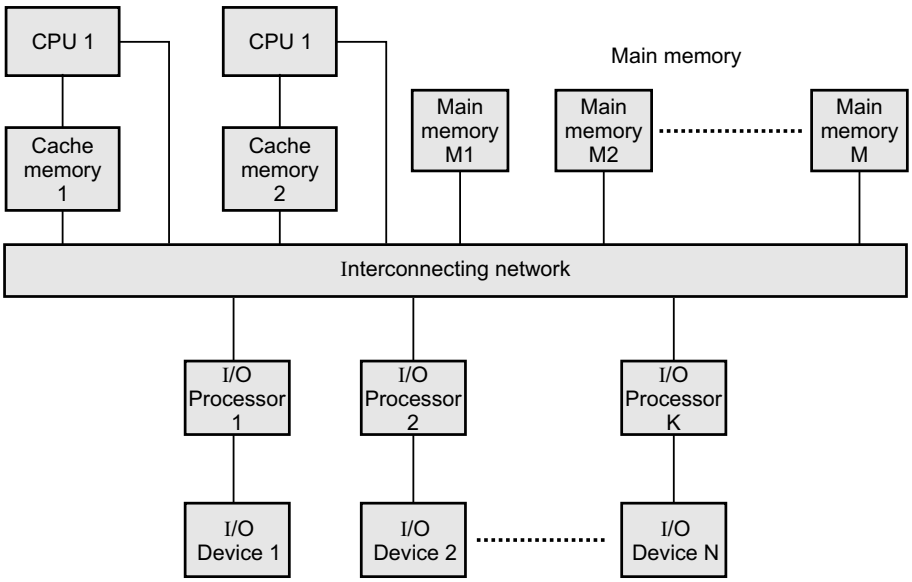


Fig. 1.6.5 Computer structure with two CPUs, main memory banks

If we link several copies of the foregoing prototype structures, more complex structures of computer can be obtained. Computer Network is an example of such a structure.

1.6.2.2 Queueing Models

In this section we will discuss an analytic performance model of a computer system. The model which is discussed here is based on queueing theory. The model which is considered here is, M/M/1 model. The first M indicates the interarrival time between two successive items requiring service from the server. The items are served in their order of arrival. i.e. First Come First Service (FCFS) scheduling. The second M indicates service time distribution. 1 indicates number of service facility centers. Fig. 1.6.6 shows simple queueing model of a computer.

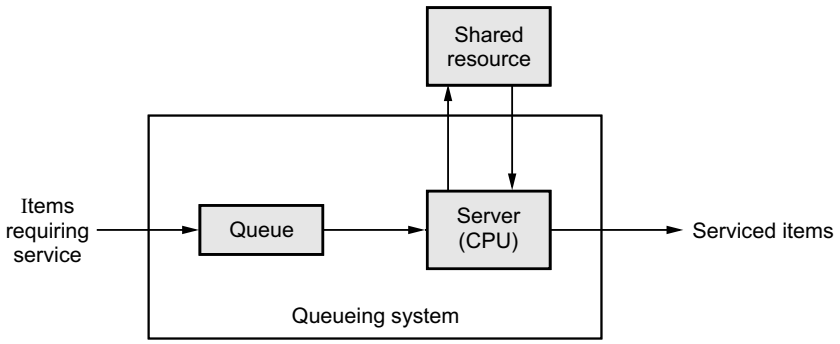


Fig. 1.6.6 Simple queueing model of a computer system

CPU is used as a server. The items or tasks requiring service by the CPU are queued in a memory. One task is processed at a time by the CPU. The tasks from the queue are processed (serviced) by the CPU on FCFS basis. The tasks requiring service and joining the queue follow probability distribution with mean (i.e. average arrival rate) denoted by λ (lambda). Also the service time distribution i.e. service rate follows probability distribution with mean denoted by μ(mu).

The traffic intensity, denoted by ρ (rho) represents the mean utilization of the server. It is given by, $\rho = \frac{\lambda}{\mu}$.

For example, if the average of two tasks are arriving per second, then $\lambda = 2$. If the average rate of servicing is eight tasks per second, then $\mu = 8$.

The traffic intensity i.e. the mean utilization of the server, $\rho = \frac{\lambda}{\mu} = \frac{2}{8} = 0.25$.

The interarrival time between two successive customers is a random variable with parameter λ. This random process is characterized by the interarrival time distribution denoted by $P_I(t)$. It is defined as the probability that at least one task arrives during a period of t. The M/M/1 case assumes that the number of items or tasks arriving or joining the queue follows Poisson distribution with parameter λ. The probability distribution is, $P_I(t) = 1 - e^{-\lambda t}$.

When t = 0, this exponential distribution has $P_I(t) = 0$. As t increases, $P_I(t)$ increases steadily toward 1 at a rate λ.

Let $P_s(t)$ be the probability that the service required by a task is completed by the CPU in time t or less after removing it from the queue. Then the probability distribution of it is given by,

$$P_s(t) = 1 - e^{-\mu t}$$

There are different performance parameters which can characterize the steady-state performance of the single-server queueing system.

1. **Traffic intensity :** It is denoted by ρ and given by $\rho = \lambda / \mu$. It is the average fraction of time the server is busy. Thus ρ is nothing but utilization of the server.
2. **Average number of tasks queued in the system :** It includes the number of tasks waiting for service and the number of tasks actually being served. It is also known as **mean queue length**. Let E(N) be the average number of tasks in the system. Then,

$$E(N) = \sum_{n=0}^{\infty} n \cdot P_n \quad \ldots (1.6.1)$$

where P_n is the probability that there are n tasks in the system. It is given by

$$P_n = (1 - \rho)\rho^n$$

Substituting in equation (i) gives,

$$E(N) = \sum_{n=0}^{\infty} n(1-\rho)\rho^n$$

$$= (1-\rho)\rho(1+2\rho+3\rho^2+4\rho^3+\cdots) = (1-\rho)\cdot\rho\frac{1}{(1-\rho)^2}$$

$$= \frac{\rho}{1-\rho}$$

$$= \frac{\lambda/\mu}{1-\lambda/\mu} \quad \ldots (\because \rho = \lambda/\mu)$$

$$= \frac{\lambda}{\mu-\lambda} \quad \ldots (1.6.2)$$

3. **Average time that tasks spend in the system :** It involves waiting time in queue and actual service time. This is also called **Average response time** or **mean waiting time**. Let E(V) be the average time that tasks spend in the system. The quantities E(V) and E(N) are related directly. When average number of tasks are E(N) and tasks enter the system at rate λ, then we can write,

$$E(V) = E(N)/\lambda \quad \ldots (1.6.3)$$

Combining equations (1.6.2) and (1.6.3), we get

$$E(V) = \frac{\lambda}{\mu-\lambda} \times \frac{1}{\lambda}$$

$$\therefore \quad E(V) = \frac{1}{\mu-\lambda}$$

4. **Average time spent waiting in the queue excluding service time :** Let it be E(W).

$$\therefore \quad E(W) = E(V) - \frac{1}{\mu}$$

where $1/\mu$ is the average time required to service a task.

$$\therefore \quad E(W) = \frac{1}{\mu-\lambda} - \frac{1}{\mu}$$

$$\therefore \quad E(W) = \frac{\lambda}{\mu(\mu-\lambda)} \quad \ldots (1.6.4)$$

5. Average number of tasks waiting in the queue excluding those being served is denoted by E(Q). The average number of tasks being serviced is λ / μ. Hence subtracting this from E(N) yields E(Q).

$$\therefore \quad E(Q) = E(N) - \rho = \frac{\lambda}{\mu - \lambda} - \frac{\lambda}{\mu}$$

$$= \frac{\lambda^2}{\mu(\mu - \lambda)} \quad \ldots (1.6.5)$$

From equation (1.6.4) and (1.6.5), we get,

$$E(W) = \frac{E(Q)}{\lambda}$$

Review Questions

1. *List the processor level components.*
2. *Write a short note on*
   a) CPU
   b) Memories
   c) IO devices
   d) Interconnection network
3. *Explain various design aspects in the processor level design.*
4. *Explain the processor level design process using prototype-structures.*
5. *Draw and explain simple queueing model of computer system.*

1.7 CPU Organization

May-09

In addition to execute programs, the CPU controls the functioning of other system components with the help of control signals. It directly or indirectly controls I/O operations such as data transfers between I/O devices and main memory. It also supports interrupt facility by which external devices can request for CPU service. The major functions of the CPU are summarized in the flowchart shown in Fig. 1.7.1.

To perform above mentioned functions CPU organization is divided into many functional unit and each functional unit is responsible for execution of particular tasks. Let us study the general CPU organization.

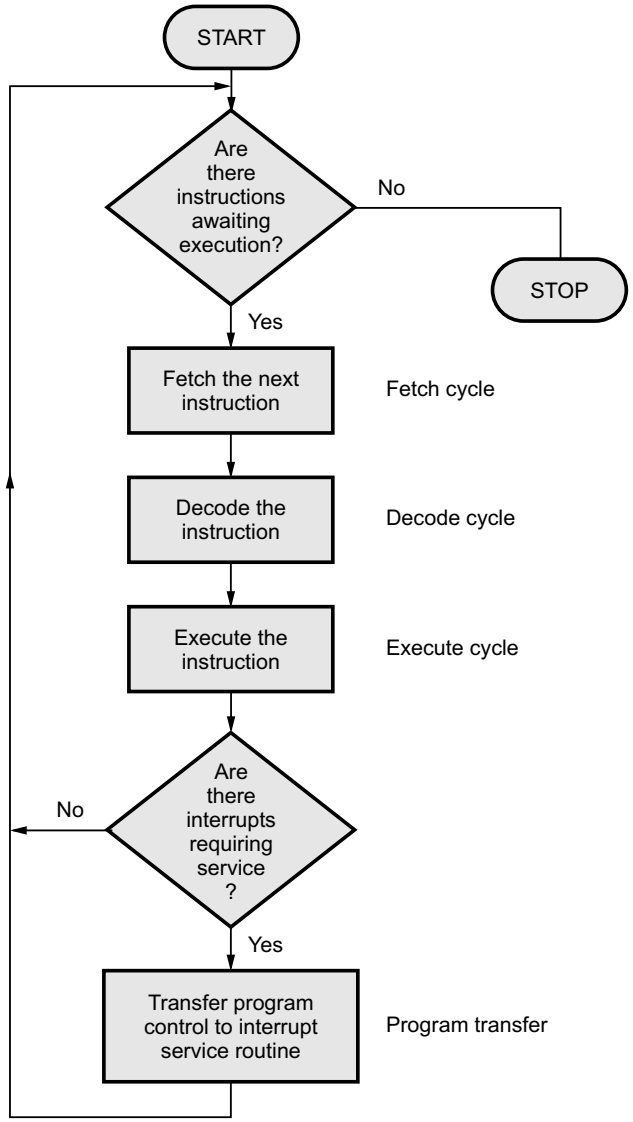


Fig. 1.7.1 Flowchart showing major functions of processor

The Fig. 1.7.2 shows the general CPU organization. It includes three major logic devices.

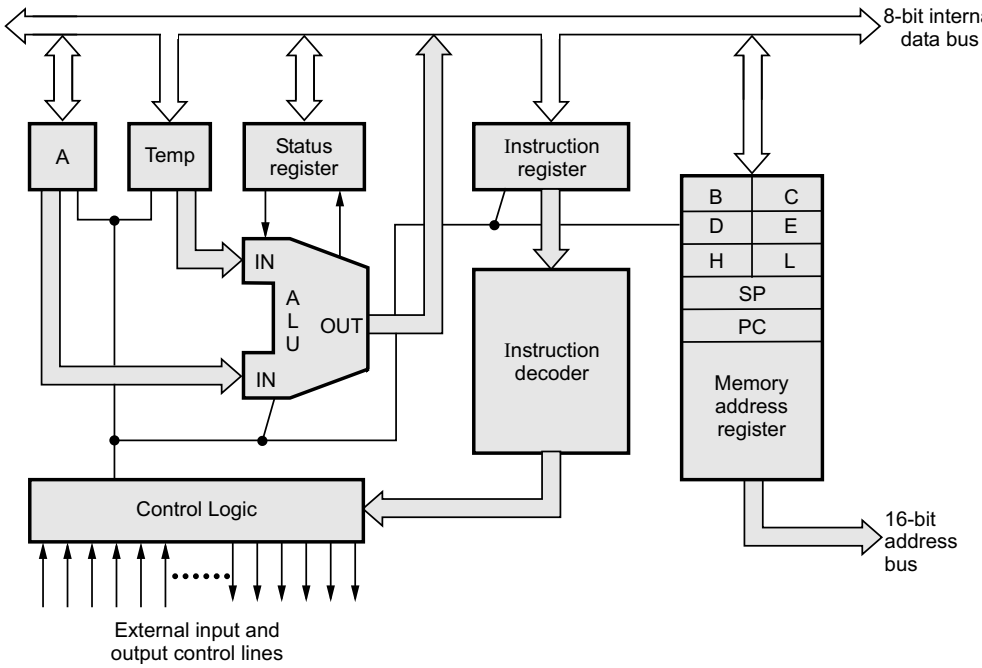


Fig. 1.7.2 General processor organization

- ALU
- Several registers
- Control unit

The internal data bus is used to transmit data between these logic devices.

ALU :

One of the CPU's major logic devices is the arithmetic logic unit (ALU). It contains the CPU's data processing logic. It has two inputs and an output. The internal data bus of CPU is connected to the two inputs of ALU through the temporary register and the accumulator.

The ALU's single output is connected to the internal data bus. It allows to send the output of ALU over the bus to any device connected to the bus. In most of the CPUs register A gives data for the ALU and after performing the operation, the resulting data word is sent to the register A and stored there. This special register, where the result is accumulated is commonly known as **accumulator**.

The ALU works on either one or two data words, depending on the kind of operation. The ALU uses input ports as necessary. For example, addition operation uses both ALU inputs while complementing data operation uses only one input. To

complement the data word, all the bits of the word's that are logic 1 are set to logic 0 and all the bits of the word at logic 0 are set to logic 1.

The ALU of the most of the CPUs can perform following functions.

- Add
- Subtract
- AND
- OR
- Exclusive OR
- Complement
- Shift right
- Shift left
- Increment
- Decrement

Registers :

Registers are a prominent part of the block diagram and the programming model of any CPU. The basic registers found in most of the CPUs are the accumulator, the program counter, the stack pointer, the status register, the general purpose registers, the memory address register, the instruction register and the temporary data registers.

Control Logic :

The control logic is a important block in the CPU. The control logic is responsible for working of all other parts of the CPU together. It maintains the synchronization in operation of different parts in the CPU. The synchronization is achieved with the help of one of the control logic's major external inputs, CPU's clock. The clock is a signal which is the basis of all the timings inside the CPU.

Usually CPU's control logic is **microprogrammed.** This means that the organization of the control logic itself is much like the organization of a very special purpose CPU.

The control logic receives the signal from instruction decoder which decodes the instruction stored in the instruction register. The control logic then generates the control signals necessary to carry out this instruction. The control logic does a few other special functions. It looks after the CPU power-up sequence. It also processes interrupts. An interrupt is like a request to the CPU from other external devices such as the memory and I/O. The interrupt asks the CPU to execute a special program.

Internal Data Bus :

The internal data bus connects the different parts of CPU together and it enables the communication between these parts. The data transfer through this internal data bus is controlled by control logic.

CPU's internal data bus usually connected to an external data bus. Due to this CPU can communicate with external memory or I/O devices. Usually the internal data bus is connected to the external data bus by logic called a bi-directional bus (transceiver).

Another way to represent CPU organization is the single bus CPU organization in which the arithmetic and logic unit and all CPU registers are connected through a single common bus. It also shows the external memory bus connected to address (AR) and data register (DR).

The registers Y, Z and Temp in Fig. 1.7.3, are used only by the CPU unit for temporary storage during the execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction. The programmer cannot access these registers. The IR and the instruction decoder are integral parts of the control circuitry in the CPU unit. All other registers and the ALU are used for storing and manipulating data. The data registers, ALU and the interconnecting bus is referred to as **data path**.

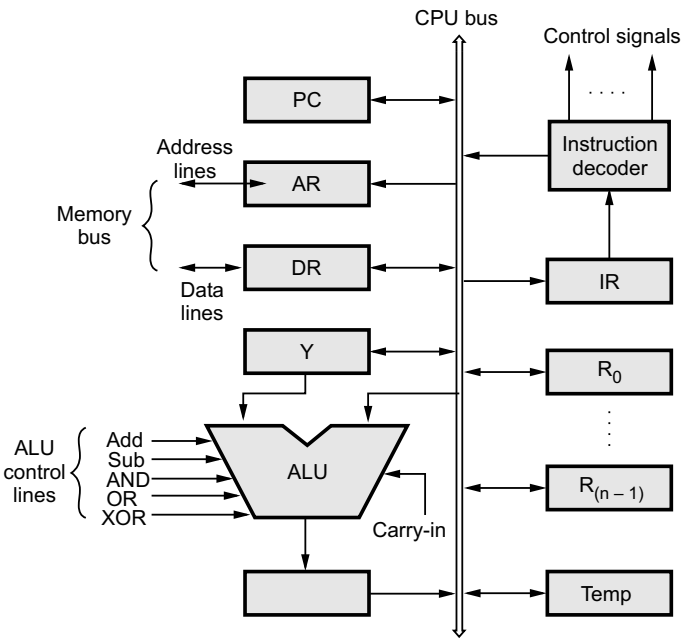


Fig. 1.7.3 Single bus organization of processor

1.7.1 CPU Register Organization

We have seen that CPU consists of various registers. These registers are used for different purposes. Let us study the functioning of these registers one by one.

The Accumulator :

The accumulator is the major working register of CPU. Most of the time it is used to hold the data for manipulation. Whenever the operation processes two words, whether arithmetically or logically, the accumulator contains one of the words. The other word may be present in another register or in a memory location. Most of the times the result of an arithmetic or logical operation is placed in the accumulator. In such cases, after execution of instruction original contents of accumulator are lost because they are overwritten.

The accumulator is also used for data transfer between an I/O port and a memory location or between one memory location and another.

The Program Counter :

The **Program Counter** is one of the most important registers in the CPU. As mentioned earlier, a program is a series of instructions stored in the memory. These instructions tell the CPU exactly how to solve a problem. It is important that these instructions must be executed in a proper order to get the correct result. This sequence of instruction execution is monitored by the program counter. It keeps track of which instruction is being used and what the next instruction will be.

The program counter gives the address of memory location from where the **next** instruction is to be fetched. Due to this the length of the program counter decides the maximum program length in bytes. For example, CPU that has 16-bit program counter, can address 2^{16} bytes (64 K) of memory.

Before the CPU can start executing a program, the program counter has to be loaded with valid memory address. This memory location must contain the opcode of first instruction in the program. In most of the CPUs this location is fixed. For example, memory address (0000H) for 16-bit program counter. The fixed address is loaded into the program counter by resetting the CPU.

As said earlier, the instructions must be executed in a proper order to get the correct result. This does not mean that every instruction must follow the last instruction in the memory. But it must follow the logical sequence of the instructions. In some situations, it is better to execute part of a program that is not in sequence (don't confuse with logical sequence) with the main program. For example, there may be a part of a program that must be repeated many times during the execution of the entire program. Rather than writing repeated part of the program again and again, the programmer can write that part only once. This part is written separately. The part of the program which is written separately is called **subroutine**. The Fig. 1.7.4 shows how the main and subroutine programs are executed.

The program counter does the major role in subroutine execution as it can be loaded with required memory address. With the help of instruction it is possible to load any memory address in the program counter. When subroutine is to be executed, the program counter is loaded with the memory address of the first instruction in the subroutine. After execution of the subroutine, the program counter is loaded with the memory address of the next instruction from where the program control was transferred to the subroutine program.

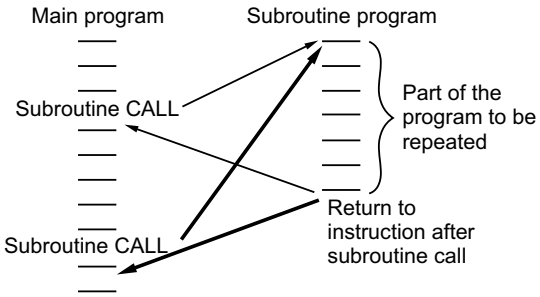


Fig. 1.7.4 Execution of subroutine programs

The Status Register : The status register is used to store the results of certain condition when certain operations are performed during execution of the program. The **status register** is also referred to as **flag register.** ALU operations and certain register operations may set or reset one or more bits in the status register. Status bits lead to a new set of CPU instructions. These instructions permit the execution of a program to change flow on the basis of the condition of bits in the status register. So the condition bits in the status register can be used to take logical decisions within the program. Some of the common status register bits are:

1) Carry/Borrow : The carry bit is set when the summation of two 8-bit numbers is greater than 1111 1111 (FFH). A borrow is generated when a large number is subtracted from a smaller number.

2) Zero : The zero bit is set when the contents of register are zero after any operation. This happens not only when you decrement the register, but also when any arithmetic or logical operation causes the contents of register to be zero.

3) Negative or sign : In 2's complement arithmetic, the most significant bit is a sign bit. If this bit is logic 1, the number is negative number, otherwise a positive number. The negative bit or sign bit is set when any arithmetic or logical operation gives a negative result.

4) Auxiliary carry : The auxiliary carry bit of status register is set when an addition in the first 4 bits causes a carry into the fifth bit. This is often referred as half carry or intermediate carry. This is used in the **BCD arithmetic**.

5) Overflow flag : In 2's complement arithmetic, most significant bit is used to represent sign and remaining bits are used to represent magnitude of a number (See Fig. 1.7.5). This flag is

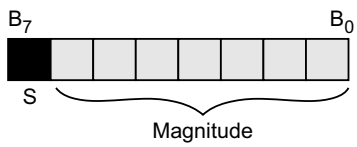


Fig. 1.7.5 2's complement 8-bit number

set if the result of a signed operation is too large to fit in the number of bits available (7-bits for 8-bit number) to represent it.

For example, if you add the 8-bit signed number 01110110 (+118 decimal) and the 8-bit signed number 00110110 (+ 54 decimal). The result will be 10101100 (+ 172 decimal), which is the correct binary result, but in this case it is too large to fit in the 7-bits allowed for the magnitude in an 8-bit signed number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

6) Parity : When the result of an operation leave the indicated register with an even number of 1s, parity bit is set.

The Stack Pointer :

This is an important register which programmer uses frequently. In the earlier sections we have seen how subroutines are executed by changing the program counter contents. But one question you may have in your mind is that how the program counter is loaded with the address of the next instruction (return address) from where the program control was transferred to the subroutine. This return address is kept in a special memory area called **the stack.** Before transferring the program control to the subroutine the return address is pushed onto the stack. After the execution of subroutine the return address is popped off from the stack and loaded into the program counter.

The memory address of the stack area is given by a special register called the **stack pointer**. Like the Program Counter, the Stack Pointer automatically points to the next available location in the memory. In most CPUs, the stack pointer decrements (points to the next lower memory address) when data is pushed on the stack. This allows the programmer to build the stack down in memory as shown in the Fig. 1.7.6. Usually stack operations are 2 byte operations. This means that the stack pointer decrements by two memory address locations each time when 2 byte data is pushed on the stack. When the data is popped off from the stack, the stack pointer is incremented by two memory address locations.

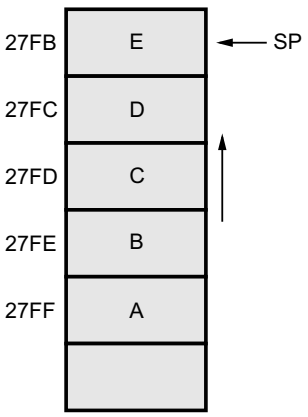


Fig. 1.7.6 Stack operation

It is important to note that as you go on storing (pushing) data on the stack, the stack pointer always points the last data placed on the stack and when you try to remove (pop) data you always get the last data placed on the stack. This kind of stack operation is called **LIFO (last in first out) operation.**

General Purpose Registers :

In addition to the six basic registers, most CPUs have other registers called **general purpose registers.** The general purpose registers are used as simple storage area, mainly these are used to store intermediate results of the operation. Getting the operand from the general purpose registers is more faster than from memory so it is better to have sufficient number of general purpose register in the CPU. The CPU used in this chapter has six general purpose registers (Refer Fig. 1.7.2) called the B, C, D, E, H, and L registers. These registers individually can operate as 8 bit registers. Together, the BC, DE, and HL registers can operate as 16-bit register pairs.

Memory Address Register :

The memory address register gives the address of memory location that the processor wants to use. That is, memory address register holds 16-bit binary number. The output of the memory address register drives the 16-bit address bus. This output is used to select a memory location.

The Instruction Register :

The instruction register holds the operation code (opcode) of the instruction the CPU is currently executing. The instruction register is loaded during the opcode fetch cycle. The contents of the instruction register is used to drive the part of the control logic known as the **instruction decoder.**

Temporary Data Register :

The need for the temporary data registers arises because the ALU has no storage of its own. The ALU has two inputs. One input is supplied by accumulator and other from temporary data register. The programmer cannot access this temporary data register and therefore it is not a part of programming model.

Review Questions

1. *Draw and explain the CPU organization.*
2. *Explain the single bus organization of processor.*
3. *Explain the CPU register organization.*
4. *Explain the use of following registers of processor.*
   a. Program counter b. Accumulator c. Instruction register d. Stack pointer
5. *Name and explain various special register in a typical computer.* **May-09, Marks 8**

1.8 Data Representation

Dec.-11, May-12

The basic form of information handled by a computer are instructions and data. The data can be in the form of numbers or nonnumerical data. The data in the number form can be further classified as fixed point and floating-point. This is illustrated in Fig. 1.8.1.

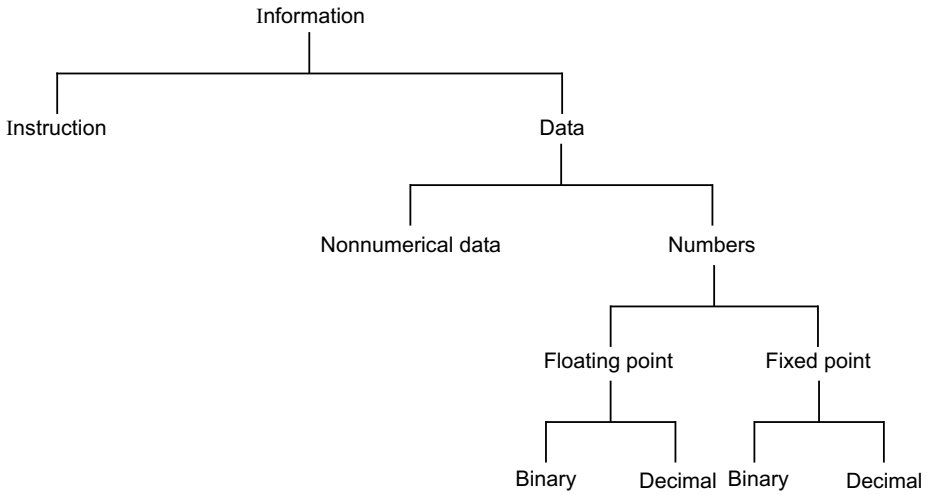


Fig. 1.8.1 The basic information types

The Fig. 1.8.1 shows that, finally, there are two ways to represent information; either in binary form or in decimal form. The digital computer represents information in binary words, where a word is a unit of information of some fixed length n. An n-bit word allows up to 2^n different items to be represented. The precision of a number word is determined by its length and therefore to represent numbers of various sizes no single word length is suitable for representing every kind of information encountered in a typical computer. Considering this fact, fixed-point numbers come in lengths of 1, 2, 4 or more bytes, whereas, floating-point numbers come in single precision (4-bytes) or double precision (8-bytes) formats.

Big-Endian and Little-Endian Assignments

There are two ways that byte addresses can be assigned across words : **big-endian** and **little-endian.** When lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word, addressing is called **big endian.** When the lower byte addresses are used for the less significant bytes (the rightmost byte) of the word, addressing is called **little-endian.** This is illustrated in Fig. 1.8.2.

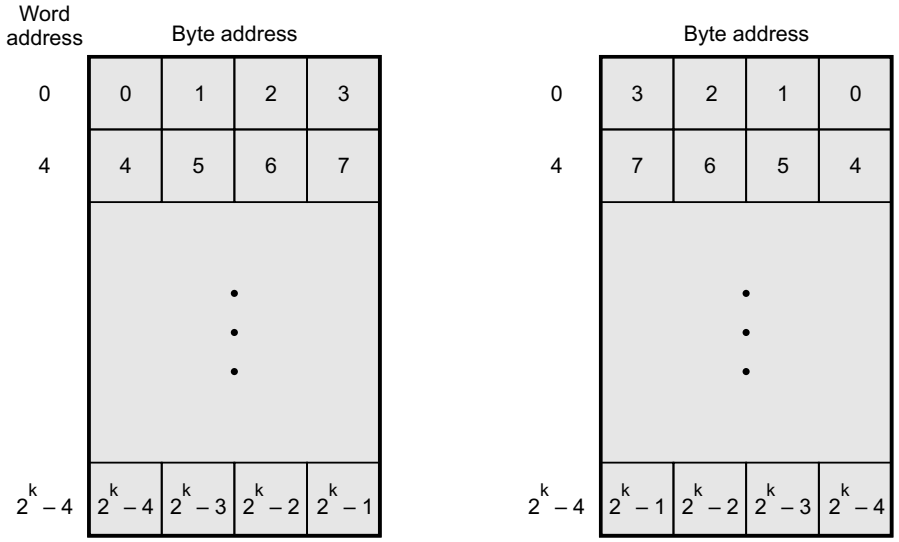


Fig. 1.8.2 Byte and word addressing

Example 1.8.1 *Consider the memory system of a computer storing the following data :*

Address in Hex	Data stored (binary)
2000	00111000
2001	00110100
2002	00110010
2003	00111001

Interpret the storage as numbers in the manner indicated below and find their decimal values in each case.

i) Big-endian storage of 2 hex words of 4-digits each.

ii) Big-endian storage of 2 BCD words of 4 -digits each.

iii) Little-endian storage, in ASCII, of a 4-digit signed hex word.

iv) Little endian storage, in ASCII, of a 4-digit BCD word.

Solution : We know that, big endian storage uses lower byte address for more-significant bytes and little-endian storage uses lower byte address for less-significant bytes. Therefore, in big-endian storage given two words are : 3834H, 3239H and in little-endian storage given two words are : 3932H, 3438H.

i) Big-endian storage of 2 hex words of 4-digit each are : 3834H, 3239H

$$3834H = 3\times16^3 + 8\times16^2 + 3\times16^1 + 4\times16^0 = 14388 \text{ decimal}$$

$$3239H = 3\times16^3 + 2\times16^2 + 3\times16^1 + 9\times16^0 = 12857 \text{ decimal}$$

ii) Big-endian storage of 2 BCD words of 4-digit each are : 3834, 3239.
3834 (BCD) = 3834 decimal

3239 (BCD) = 3239 decimal

iii) Little-endian storage in ASCII, of 4-digit signed hex word is : 9248H. **Note :** We subtract 30H from ASCII code to get the digit value in hex.
$9248H = 9\times16^3 + 2\times16^2 + 4\times16^1 + 8\times16^0 = 37448$ decimal

iv) Little-endian storage, in ASCII, of a 4-digit BCD word is : 9248
9248 BCD = 9248 decimal

Error detection and correction

When the digital information in the binary form is transmitted from one circuit or system to another circuit or system an error may occur. This means a signal corresponding to 0 may change to 1 or vice-versa due to presence of noise. To maintain the data integrity between transmitter and receiver, extra bit or more than one bit are added in the data. These extra bits allow the detection and some times correction of error in the data. The data along with the extra bit/bits forms the code. Codes which allow only error detection are called **error detecting codes** and codes which allow error detection and correction are called **error detecting and correcting codes**.

A parity bit is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1s either odd or even. The message, including the parity bit is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a **parity generator** and the circuit that checks the parity in the receiver is called a **parity checker**.

In even parity the added parity bit will make the total number of 1s an even amount. In odd parity the added parity bit will make the total number of 1s an odd amount.

In the even parity we can obtained the parity bit from following expression :

$P = D_0 \oplus D_1 \oplus D_2 \ldots \oplus D_n$ where D indicates data bits and $\oplus$ sign indicates exclusive-OR operation. In case of odd parity, the expression is

$$\overline{P} = D_0 \oplus D_1 \oplus D_2 \ldots \oplus D_n$$

As a general rule in a digital system where the transmission system is relatively short, it may be assumed that the probability of a single-bit error is small and that of a double-bit error and higher order errors is extremely small. The parity error detection system just described detects any odd number of errors. However, it cannot detect an even number of error because such errors will not destroy the parity of the transmitted group of bits.

Some codes such as Hamming code not only provides the detection of a bit error, but also identifies which bit is in error so that it can be corrected. Thus Hamming code is called error detecting and correcting code. The code uses a number of parity bits (dependent on the number of information bits) located at certain positions in the code group.

For single error correction and detection we can determine the number of parity bits needed for particular number of information bits. If number of information bits are x, then the number of parity bits, P is determined by the following relationship :

$$2^P \geq x + P + 1 \quad \ldots (1.8.1)$$

For example, if we have four information bit, i.e. x = 4, then P is found by trial and error using equation 1. Let P = 2. Then

$$2^P = 2^2 = 4$$

and

$$x + P + 1 = 4 + 2 + 1 = 7$$

Since 2^P must be equal to or greater than x + P + 1, the relationship in equation 1 is not satisfied. Hence we have to try with next value of P. Let P = 3.

Then

$$2^P = 2^3 = 8$$

and

$$x + P + 1 = 4 + 3 + 1 = 8$$

This value of P satisfies the relationship given is equation 1, and therefore we can say that three parity bits are required to provide single error correction for four information bits.

A Hamming code provides detection and correction of a single error. However, by adding one more parity bit in the Hamming code, it is possible detect double error. Such code is known as single error correcting and double error detecting (SECDED) code. The Fig. 1.8.3 shows the structure of a typical error detection and correction scheme used in computer system.

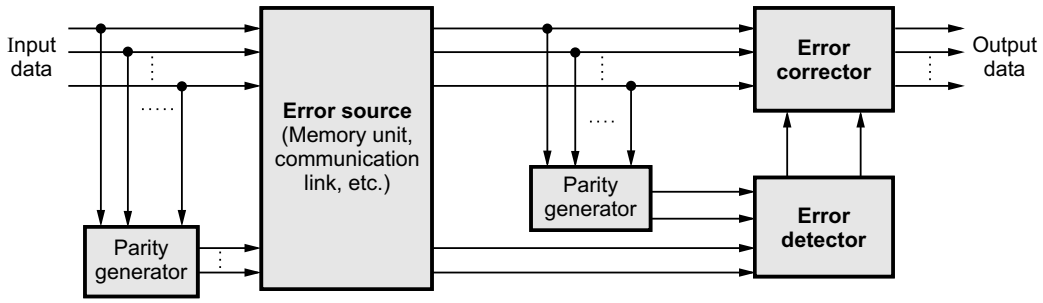


Fig. 1.8.3 Error detection and correction system

In binary number system, a number can be represented as an integer or a fraction. Depending on the design, the hardware can interpret number as an integer or fraction.

The radix point is never explicitly specified. It is implicited in the design and the hardware interprets it accordingly. In integer numbers, radix point is fixed and assumed to be to the right of the right most digit. As radix point is fixed, the number system is referred to as **fixed point number system**.

With fixed point number system we can represent positive or negative integer numbers. However, floating point number system allows the representation of numbers having both integer part and fractional part.

Number system is a basis for counting various items. On hearing the word 'number', all of us immediately think of the familiar decimal number system with its 10 digits : 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Modern computers communicate and operate with binary numbers which use only the digits 0 and 1. Let us consider decimal number 18. This number is represented in binary as 10010. In the example, if decimal number is considered, we require only two digits to represent the number, whereas if binary number is considered we require five digits. Therefore we can say that, when decimal quantities are represented in the binary form, they take more digits. For large decimal numbers people have to deal with very large binary strings and therefore, they do not like working with binary numbers. This fact gave rise to three new number systems : Octal, Hexadecimal and Binary Coded Decimal (BCD). These number systems represent binary number in a compressed form. Therefore, these number systems are now widely used to compress long strings of binary numbers.

1.8.1 Decimal Number System

Before considering any number system, let us consider familiar decimal number system. In decimal number system we can express any decimal number in units, tens, hundreds, thousands and so on. When we write a decimal number say, 5678.9, we know it can be represented as,

$5000 + 600 + 70 + 8 + 0.9 = 5678.9$

The decimal number 5678.9 can also be written as 5678.9_{10}, where the 10 subscript indicates the **radix or base.**

This says that, the position of a digit with reference to the decimal point determines its value/weight. The sum of all the digits multiplied by their weights gives the total number being represented. The leftmost digit, which has the greatest weight is called the **most significant digit** and the rightmost digit, which has the least weight, is called the **least significant digit**. Fig. 1.8.4 shows decimal digit and its weights expressed as a power of 10.

	10^3	10^2	10^1	10^0		10^{-1}
	5	**6**	**7**	**8**	**.**	**9**
In power of 10	5×10^3	6×10^2	7×10^1	8×10^0	.	9×10^{-1}
	MSD					**LSD**

Fig. 1.8.4

1.8.2 Binary Number System

We know that decimal system with its ten digits is a base-ten system. Similarly, binary system with its two digits is a **base-two system**. The two binary digits (bits) are 1 and 0. Like digital system, in binary system each binary digit commonly known as bit, has its own value or weight.

In binary system, weight is expressed as a power of 2. The Fig. 1.8.5 (a) shows representation of binary number 1101.101 in power of 2. By adding each digit of a binary number in a power of 2 we can find the decimal equivalent of the given binary number.

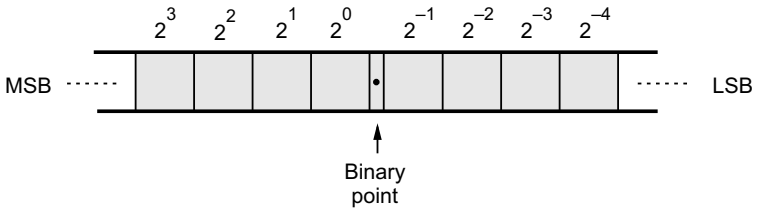


Fig. 1.8.5 Binary position values as a power of 2

	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
N =	**1**	**1**	**0**	**1**	•	**1**	**0**	**1**
	1×2^3	1×2^2	1×2^1	1×2^0	•	1×2^{-1}	1×2^{-2}	1×2^{-3}

$$N = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = (13.625)_{10}$$

Fig. 1.8.5 (a)

1.8.3 Octal Number System

We know that the base of the decimal number system is 10 because it uses the digits 0 to 9 and the base of binary number system is 2 because it uses digits 0 and 1. The octal number system uses first eight digits of decimal number system : 0, 1, 2, 3, 4, 5, 6 and 7. As it uses 8 digits, its base is 8.

For example, the octal number 5632.471 can be represented in power of 8 as shown in Fig. 1.8.5 (b). By adding each digit of an octal number in a power of 8 we can find the decimal equivalent of the given octal number.

	8^3	8^2	8^1	8^0		8^{-1}	8^{-2}	8^{-3}
N =	**5**	**6**	**3**	**2**	•	**4**	**7**	**1**
	5×8^3	6×8^2	3×8^1	2×8^0	•	4×8^{-1}	7×8^{-2}	1×8^{-3}

$$N = 5 \times 8^3 + 6 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 + 4 \times 8^{-1} + 7 \times 8^{-2} + 1 \times 8^{-3} = (2970.611328)_{10}$$

Fig. 1.8.5 (b)

1.8.4 Hexadecimal Number System

The hexadecimal number system has a base of 16 having 16 characters : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. It is another number system that is particularly useful for human communications with a computer. Although it is somewhat more difficult to interpret than the octal number system, it has become the most popular. Since its base is a power of 2 (2^4), it is easy to convert hexadecimal numbers to binary and vice versa.

Table 1.8.1 shows the relationship between decimal, binary, octal and hexadecimal. Note that each hexadecimal digit represents a group of four binary digits, called **nibbles**, that are fundamental parts of larger binary words.

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Table 1.8.1 Relation between decimal, binary, octal and hexadecimal numbers

For example, 3FD.84 can be represented in power of 16 as shown below. By adding each digit of a hexadecimal number in a power of 16 we can find decimal equivalent of the given hexadecimal number.

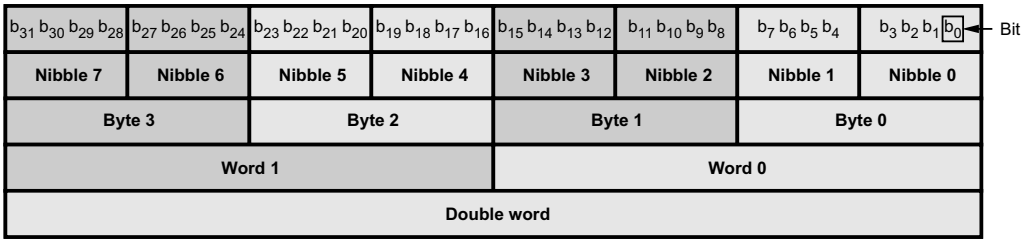
	16^2	16^1	16^0		16^{-1}	16^{-2}
N =	**3**	**F**	**D**	**•**	**8**	**4**
	3×16^2	$F \times 16^1$	$D \times 16^0$	•	8×16^{-1}	4×16^{-2}

$$N = 3 \times 16^2 + F \times 16^1 + D \times 16^0 + 8 \times 16^{-1} + 4 \times 16^{-2}$$

$$= 3 \times 16^2 + 15 \times 16^1 + 13 \times 16^0 + 8 \times 16^{-1} + 4 \times 16^{-2} = \mathbf{(1021.515625)_{10}}$$

1.8.5 Format of a Binary Number

The following figure shows the format of binary number. Four binary digits form a **nibble**, eight binary digits form a **byte**, sixteen binary digits form a word and thirty-two binary digits form a double-word.



Nibble : 4-bits can represent 2^4 = 16 distinct values

Byte : 8-bits can represent 2^8 = 256 distinct values

Word : 16-bits can represent 2^{16} = 65536 distinct values

Double word : 32-bits can represent 2^{32} = 4294967296 distinct values.

Example 1.8.2 *Convert the following binary number to decimal number.*

a) 110101_2 *b)* 111100_2 **May-09, Marks 2**

Solution :

$$(110101)_2 = 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$$
$$= 32 + 16 + 0 + 4 + 0 + 1 = (53)_{10}$$
$$(111100)_2 = 1\times 2^5 + 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 0\times 2^0$$
$$= 32 + 16 + 8 + 4 + 0 + 0 = (60)_{10}$$

Review Questions

1. *Explain basic information types.*
2. *Write a short note on number system.*
3. *How does one detect and correct errors during data transmission ?* **Dec.-11, Marks 8**
4. *Draw and explain the different types of data representations.* **May-12, Marks 8**

1.9 Fixed Point Numbers **Dec.-07, May-09**

In binary number system, a number can be represented as an integer or a fraction. Depending on the design, the hardware can interpret number as an integer or fraction. The radix point is never explicitly specified. It is implicited in the design and the hardware interprets it accordingly. In integer numbers, radix point is fixed and assumed to be to the right of the right most digit. As radix point is fixed, the number system is referred to as *fixed point number system*.

The fixed point/integer numbers are represented in two forms : signed integer and unsigned integer. Unsigned integer numbers represent positive numbers. To represent negative numbers various techniques are used because computer does not have provision to represent negative sign. In this section we will see techniques to represent signed integer numbers :

- Sign-magnitude representation
- 1's complement
- 2's complement

1.9.1 Signed Binary Numbers

In practice, we use plus sign to represent positive number and minus sign to represent negative number. However, because of hardware limitations, in computers, both positive and negative numbers are represented with only binary digits. The leftmost bit (sign bit) in the number represents sign of the number. The sign bit is 0 for positive numbers and it is 1 for negative numbers. These numbers are represented by the signed magnitude format. Fig. 1.9.1 shows the sign magnitude format for 8-bit signed number.

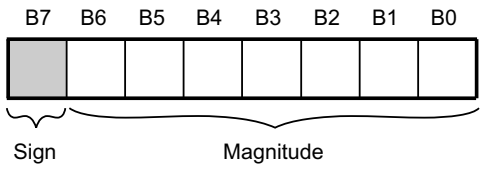


Fig. 1.9.1

Here, the most significant bit (MSB) represents sign of the number. If MSB is 1, number is negative and if MSB is 0, number is positive. The remaining bits represent magnitude of the number. Here are some examples of sign-magnitude numbers.

1. + 6 = 0 0 0 0 0 1 1 0
2. – 14 = 1 0 0 0 1 1 0 0
3. + 24 = 0 0 0 1 1 0 0 0
4. – 64 = 1 1 0 0 0 0 0 0

In case of unsigned 8-bit binary numbers the decimal range is 0 to 255. For signed magnitude 8-bit binary numbers the largest magnitude is reduced from 255 to 127 because we need to represent both positive and negative numbers.

Maximum positive number	0111 1111	= +127
Maximum negative number	1111 1111	= –128

Drawbacks of sign-magnitude representation

For addition and subtraction, it is necessary to consider signs of both the numbers and their relative magnitudes in order to carry out the required arithmetic operation.

There are two representations of zeros :

$+0_{10}$ = 0 0 0 0 0 0 0 0

-0_{10} = 1 0 0 0 0 0 0 0

Due to the two representations of zero, it is more difficult to *test for zero operation frequently performed by the computer.*

1.9.2 1's Complement Representation

The 1′s complement of a binary number is the number that results when we change all 1′s to zeros and the zeros to ones.

Example 1.9.1 *Find 1's complement of* $(11010100)_2$.

Solution :

1	1	0	1	0	1	0	0	**Number**
▽	▽	▽	▽	▽	▽	▽	▽	**NOT operation**
0	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1's complement of number**

1.9.3 2's Complement Representation

The 2′s complement is the binary number that results when we add 1 to the 1′s complement. It is given as

2′s complement = 1′s complement + 1

The 2′s complement form is used to represent negative numbers.

Example 1.9.2 *Find 2's complement of* $(11000100)_2$.

Solution : Let us see how – 6 is represented in these three formats. Consider the number 6 represented in binary with eight bits.

Signed-magnitude representation : 10000110

1	1	0	0	0	1	0	0	**Number**
					1	**1**		**Carry**
0	0	1	1	1	0	1	1	**1's complement of number**
+							1	**Add 1**
0	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**2's complement of number**

Signed-1's complement representation : 11111001

Signed-2's complement representation : 11111010

In signed-magnitude, – 6 is obtained from + 6 by changing the sign bit in the leftmost position from 0 to 1. In signed-1's complement, – 6 is obtained by complementing all the bits of +6, including the sign bit. The signed 2's complement representation of – 6 is obtained by taking 2's complement of positive number, including the sign bit.

Decimal	Signed 2's complement	Signed 1's complement	Signed magnitude
+ 7	0111	0111	0111
+ 6	0110	0110	0110
+ 5	0101	0101	0101
+ 4	0100	0100	0100
+ 3	0011	0011	0011
+ 2	0010	0010	0010
+ 1	0001	0001	0001
+ 0	0000	0000	0000
– 0	–	1111	1000
– 1	1111	1110	1001
– 2	1110	1101	1010
– 3	1101	1100	1011
– 4	1100	1011	1100
– 5	1011	1010	1101
– 6	1010	1001	1110
– 7	1001	1000	1111
– 8	1000	–	–

Table 1.9.1

The Table 1.9.1 lists all possible 4-bit signed binary numbers in the three representations. Looking at the Table 1.11 we understand following points :

- Positive numbers in all three representations are identical and have 0 in the leftmost position.
- All negative numbers have a 1 in the leftmost bit position.
- The signed 2's complement system has only one representation for 0, which is always positive.
- The signed-magnitude and 1's complement systems have either a positive 0 or a negative 0.
- With four bits, we can represent 16 binary numbers.

The sign-magnitude representation requires separate handling for sign and magnitude during arithmetic operations and hence it is suitable in computer arithmetic. Therefore,

the signed complement numbers are normally used in computer arithmetic. The 1's complement imposes some difficulties and is seldom used for arithmetic operations. It is used as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The signed-2's complement system is commonly used in computer arithmetic.

1.9.4 Sign Extension

We have seen that how to represent binary number in 1's complement and 2's complement form. In this representation it is important to see how such numbers are represented.

When we add two binary numbers the result may extend by 1-bit i.e. the resulted binary number may need one more bit if there is a carry after addition of most significant bit. To represent such result in correct format we have to allocate the additional bit for representing the magnitude of the numbers which are to be added. This is illustrated in Fig. 1.9.2.

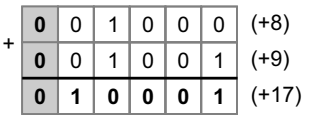


Fig. 1.9.2

In case of positive binary number we extend the bit by appending bit-0 (sign-bit for positive number) to the left of the most-significant bit of the number. In case of negative binary number can extend the bit by appending bit-1 (sign-bit for negative number) to the left of the most-significant bit of the number. This is illustrated in the Fig. 1.9.3.

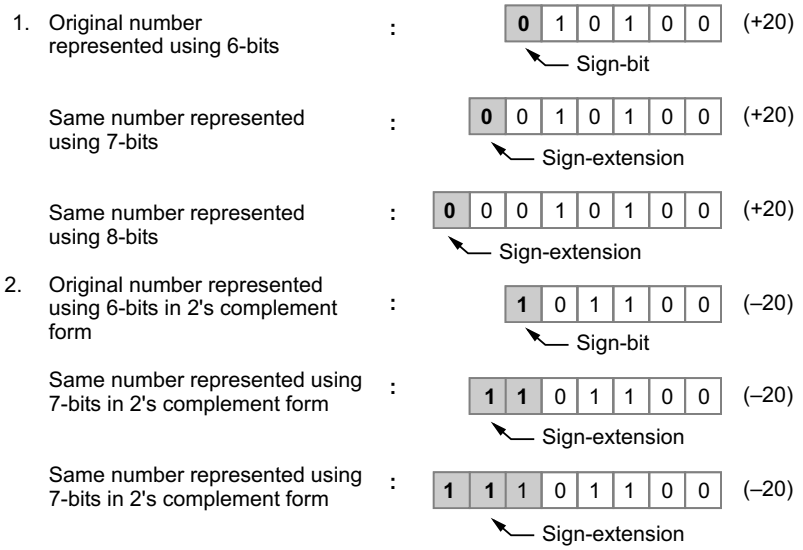

1. Original number represented using 6-bits : 0 1 0 1 0 0 (+20)
Sign-bit
Same number represented using 7-bits : 0 0 1 0 1 0 0 (+20)
Sign-extension
Same number represented using 8-bits : 0 0 0 1 0 1 0 0 (+20)
Sign-extension
2. Original number represented using 6-bits in 2's complement form : 1 0 1 1 0 0 (–20)
Sign-bit
Same number represented using 7-bits in 2's complement form : 1 1 0 1 1 0 0 (–20)
Sign-extension
Same number represented using 7-bits in 2's complement form : 1 1 1 0 1 1 0 0 (–20)
Sign-extension

Fig. 1.9.3

In both the cases sign-bit of the original number is extended and hence such extension is known as sign-extension of the number.

Review Questions

1. *What do you mean by fixed point numbers ?*
2. *Assuming 8-bit word length, express the following decimal numbers in :*
   i) Sign magnitude ii) One's complement and
   iii) Two's complement form a) –39 b) –120 **(Ans. : a) 10100111 ii) 11011000 iii) 1101 1001 b) ii) 1000 0111 iii) 1000 1000)**
3. *List the advantages of 2's complement representation over 1's complement and sign-magnitude number representation.*
4. *Solve the following using 2's complement arithmetic.*
   a) Add 46 and 18 b) Subtract 34 from 52
5. *How are the signed binary numbers represented ?* **May-09, Marks 2**
6. *What are the various ways of representing signed integers in the system ?* **Dec.-07, Marks 2**

1.10 Floating Point Numbers

Dec.-06, May-07, 08

So far, we have discussed numbers without any decimal point, fixed point numbers. The decimal point is always assumed to be to the right of the least significant digit. To represent fractional binary numbers it is necessary to consider binary point. If binary point is assumed to the right of the sign bit we can represent fractional binary number as given below.

$$B = (b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots\ldots + b_{-(n-1)} \times 2^{-(n-1)})$$

With this fractional number system we can represent the fractional numbers in the following range :

$$2^{-(n-1)} \leq F \leq 1 - 2^{-(n-1)}$$

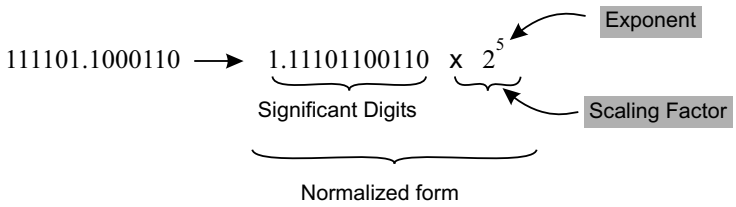
If n = 32, then value range is approximately

$$2^{(-31)} \leq F \leq 1 - 2^{-(31)} \ (1 - 2.3283 \times 10^{-10})$$

But this range is not sufficient to represent fractional numbers. To accommodate very large integers and very small fractions, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In this case, the binary point is said to **float, and the numbers are called floating-point numbers**.

We know that, the position of binary point in floating point numbers is variable and hence numbers must be represented in the specific manner referred to as **floating point representation**.

The floating point representation has three fields : **sign**, **significant digits** and **exponent**. Let us consider the number 1 1 1 1 0 1 . 1 0 0 0 1 1 0 to be represented in the floating point format. To represent the number in floating point format, first binary point is shifted to right of the first bit and the number is multiplied by the correct scaling factor to get the same value. The number is said to be in the **normalized form** and is given as



It is important to note that the base in the scaling factor is fixed, 2 and therefore, it does not need to appear explicitly in the machine representation of a floating point number. So finally, three fields sign, significant digits and **exponent represent** floating point number system. The string of the significant digits is commonly known as **mantissa**.

In the above example, we can say that,

Sign = 0

Mantissa = 1 1 1 0 1 1 0 0 1 1 0

Exponent = 5

In floating point numbers, bias value is added to the true exponent. This solves the problem of representation of negative exponent. Due to this the magnitude of two numbers can be compared by doing arithmetic on the exponent first.

1.10.1 IEEE Standard for Floating-Point Numbers

The standards for representing floating point numbers in 32-bits and 64-bits have been developed by the Institute of Electrical and Electronics Engineers (IEEE), referred to as **IEEE 754 standards**. Fig. 1.10.1 shows these IEEE standard formats.

The 32-bit standard representation shown in Fig. 1.10.1 (a) is called a **single-precision representation** because it occupies a single 32-bit word. The 32-bits are divided into three fields as shown below :

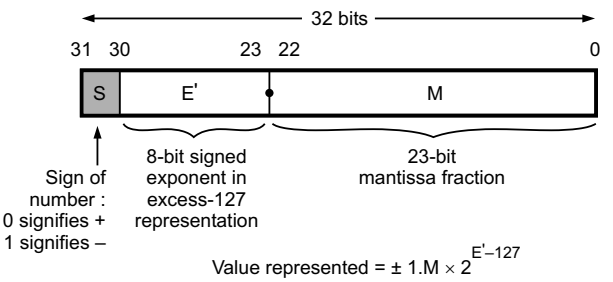

32 bits
31 30 23 22 0
S
E'
M
Sign of number : 0 signifies + 1 signifies –
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
Value represented = ± 1.M × 2^(E'–127)

Fig. 1.10.1 (a) Single-precision

(field 1) Sign ← 1 - bit

(field 2) Exponent ← 8 - bits

(field 3) Mantissa ← 23 - bits

The sign of the number is given in the first bit, followed by a representation for the exponent (to the base 2) of the scale factor. Instead of the signed exponent, E, the value actually stored in the exponent field is E′ = E + bias. In the 32-bit floating point system (single precision), bias is 127. Hence E′ = E + 127. This representation of exponent is also called as the *excess-127 format.* The end values of E′ , namely, 0 and 255, are used to indicate the floating point values of exact zero and infinity, respectively in single precision. Thus range of E′ for normal values in single precision is 0 < E′ < 255. This means that for 32-bit representation the actual exponent E is in the range – 126 ≤ E ≤ 127.

The 64-bit standard representation shown in Fig. 1.10.1 (b) is called a **double-precision representation** because it occupies two 32-bit words. The 64-bits are divided into three fields as shown below :

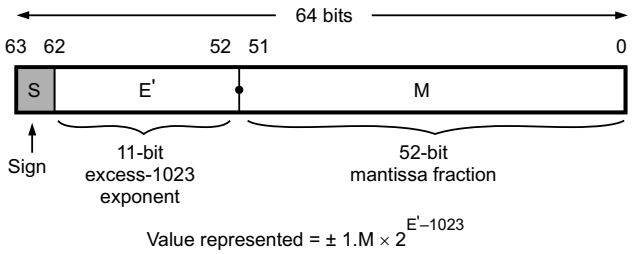


Fig. 1.10.1 (b) Double precision

(field 1) Sign ← 1 - bit

(field 2) Exponent ← 11 - bit

(field 3) Mantissa ← 52 - bits

In the double precision format value actually stored in the exponent field is given as

$$E' = E + 1023$$

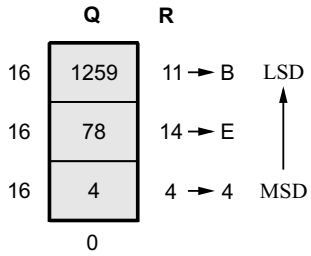
Here, bias value is 1023 and hence it is also called **excess-1023 format**. The end values of E′ , namely, 0 and 2047, are used to indicate the floating point exact values of exact zero and infinity, respectively. Thus the range of E′ for normal values in double precision is 0 < E′ < 2047. This means that for 64-bit representation the actual exponent E is in the range

$$-1022 \leq E \leq 1023.$$

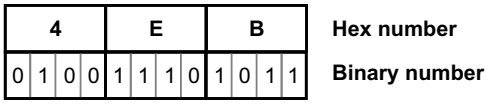
Example 1.10.1 *Represent 1259.125_{10} in single precision and double precision formats*

Solution : Step 1 : Convert decimal number in binary format

Integer part :

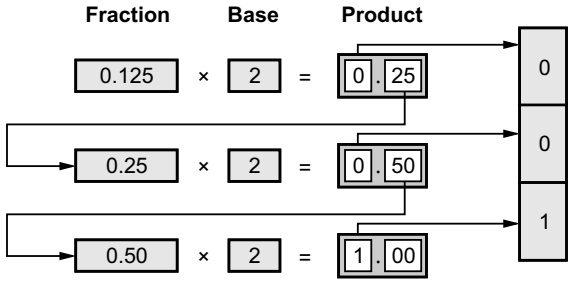


$$\therefore (1259)_{10} = (4EB)_{16}$$



$$\therefore (1259)_{10} = (100\ 1110\ 1011)_2$$

Fractional part :



$$\therefore (0.125)_{10} = (0.001)_2$$

$$\therefore \text{Binary number} = 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1 + 0.\ 0\ 0\ 1$$

$$= 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ .\ 0\ 0\ 1$$

Step 2 : Normalize the number

$$1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1.\ 0\ 0\ 1 = 1.0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1 \times 2^{10}$$

Now we will see the representation of the numbers in single precision and double precision formats.

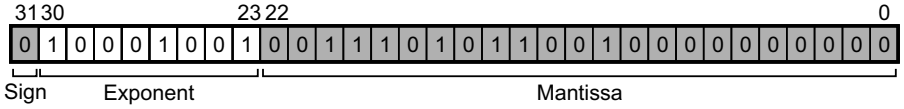
Step 3 : Single precision representation

For a given number S = 0, E = 10 and M = 0 0 1 1 1 0 1 0 1 1 0 0 1

Bias for single precision format is = 127

$\therefore \quad E' = E + 127 = 10 + 127 = 137_{10} = 1\,0\,0\,0\,1\,0\,0\,1_2$

$\therefore$ Number in single precision format is given as



Step 4 : Double precision representation

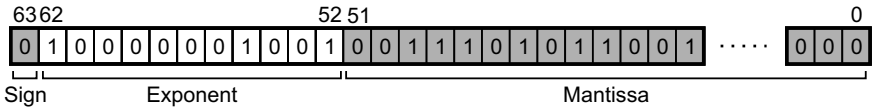
For a given number

S = 0, E = 10, and M = 0 0 1 1 1 0 1 0 1 1 0 0 1

Bias for double precision format is = 1023

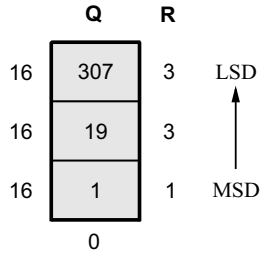
$\therefore \quad E' = E + 1023 = 10 + 1023 = 1033_{10} = 1\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1_2$

$\therefore$ Number in double precision format is given as

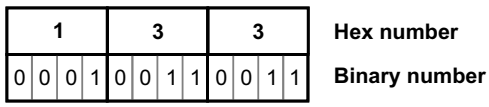


Example 1.10.2 *Represent* -307.1875_{10} *in single precision and double precision formats.*

Solution : Step 1 : Convert decimal number in binary format integer part

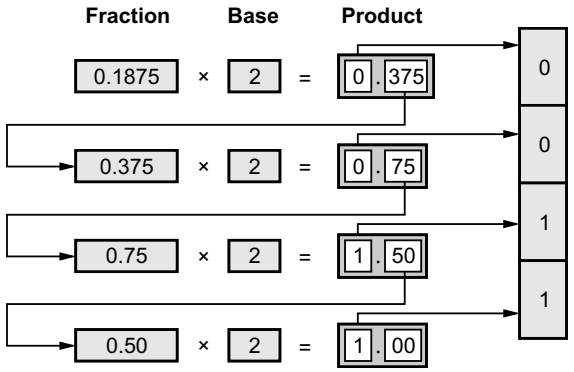


Integer part :



$\therefore \quad (307)_{10} = (133)_{16}$

$\therefore \quad (307)_{10} = (100110011)_2$



Fractional part :

$\therefore \quad (0.1875)_{10} = (0.0011)_2$

$\therefore$ Binary number $= -1\,0\,0\,1\,1\,0\,0\,1\,1 + .\,0\,0\,1\,1$

$= -1\,0\,0\,1\,1\,0\,0\,1\,1 \quad .\,0\,0\,1\,1$

Step 2 : Normalize the number

$$-1\,0\,0\,1\,1\,0\,0\,1\,1\,.\,0\,0\,1\,1 = -1.0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1 \times 2^8$$

Now we will see the representation of the numbers in single precision and double precision formats.

Step 3 : Single precision representation

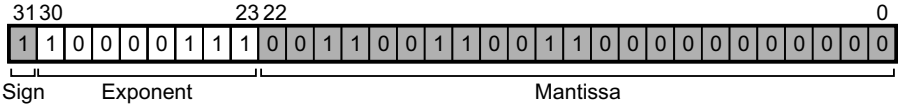
For a given number

S = 1, E = 8, and M = 0011 0011 0011

Bias for single precision format is = 127

$\therefore \quad E' = E + 127 = 8 + 127 = 135_{10} = 10000111_2$

$\therefore$ Number in single precision format is given as



Step 4 : Double precision representation

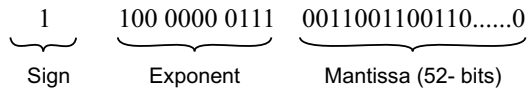
For a given number

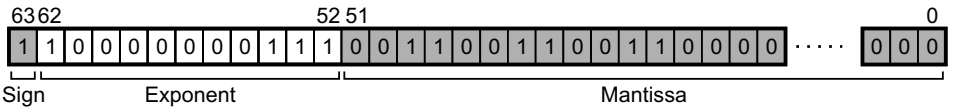
S = 1, E = 8 and M = 0011 0011 0011

Bias for double precision format is = 1023

$\therefore \quad E' = E + 1023 = 8 + 1023 = 1031_{10} = 10000000111_2$

∴ Number in double precision format is given as



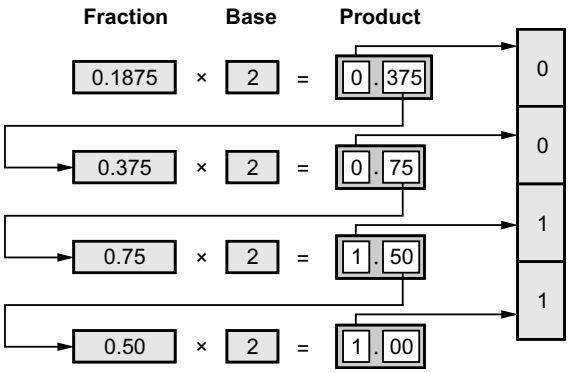


Example 1.10.3 *Represent* 0.0625_{10} *in single precision and double precision formats.*

Solution : Step 1 : Convert decimal number in binary format

Integer part : 0

Fractional part :



$$(0.0625)_{10} = (0.0001)_2$$

∴ Binary number = 0.0001

Step 2 : Normalize the number

$$0.0001 = 1.0 \times 2^{-4}$$

Now we will see the representation of the numbers in single precision and double precision formats

Step 3 : Single precision representation

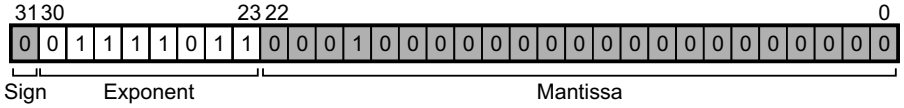
For a given number

S = 0, E = – 4, and M = 0001

Bias for single precision format is = 127

$\therefore$ $E' = E + 127 = -4 + 127 = 123_{10} = 01111011_2$

$\therefore$ Number in single precision format is given as



Step 4 : Double precision representation

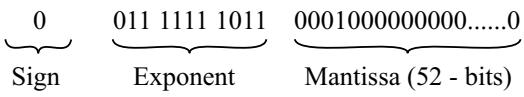
For a given number

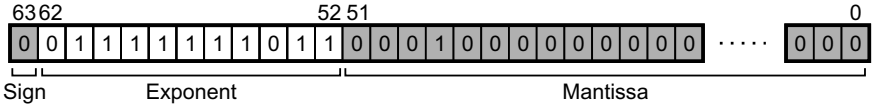
$S = 0, \quad E = -4, \quad$ and $M = 0001$

Bias for double precision format is = 1023

$\therefore$ $E' = E + 1023 = -4 + 1023 = 1019 = 01111111011_2$

$\therefore$ Number in double precision format is given as





1.10.2 Special Values

The extreme values of biased exponent, E′ are used to represent special values. When E′ = 0 and the mantissa fraction M is zero, the value exact 0 is represented. The biased exponent with all 1s is reversed to represent infinity, where infinity is the result of dividing a normal number by zero. The sign bit is still part of these representations, so there are ± 0 and $\pm \infty$ representations.

When E′ = 0 and $M \neq 0$, **denormal** numbers are represented. A denormal is a result that causes an underflow and has leading 0s in the mantissa even after the exponent is adjusted to its smallest possible value. The E′ with all 1s and $M \neq 0$ is reserved to represent Not a Number (NaN). A NaN is the result of performing an invalid operation such as 0/0 or $\sqrt{-1}$.

1.10.3 Exceptions

According to IEEE standards, the processor sets exception flags if underflow, overflow, divide by zero, inexact or invalid conditions occur during the program execution.

Underflow : In a single precision, if the number requires an exponent less than – 126 or in a double precision, if the number requires an exponent less than – 1022 to represent its normalized form the underflow occurs.

Overflow : In a single precision, if the number requires an exponent greater than + 127 or in a double precision, if the number requires an exponent greater than + 1023 to represent its normalized form the overflow occurs.

Divide by zero : Divide by zero exception occurs when any number is divided by zero.

Inexact : Inexact is the name for a result that requires rounding in order to be represented in one of the normal formats.

Invalid : An invalid exception occurs if operations such as $0/0$ or $\sqrt{1}$ are attempted. When exception occurs, the results are set to special values. System and user defined routines are used to handle such exceptions.

Review Questions

1. *In conforming to the IEEE standard mention any four situations under which a processor sets exception flag.* **Dec.-06, Marks 2**
2. *Define underflow and overflow.* **May-08, Marks 2**
3. *Describe the IEEE standards for single precision and double precision floating point numbers.*
4. *Represent following numbers in single precision and double precision formats.*
   a) 1365.125_{10} *b)* 0.625_{10} *c)* $-450.7.250_{10}$
5. *Give the IEEE standard double precision floating point format.* **Dec.-06, Marks 3**
6. *Explain the representations of floating point numbers in detail.* **May-07, Marks 6**

1.11 Instruction Formats

Dec.-03, 06, 07, 10, May-06, 08, 09

Computers may have instructions of several different lengths containing varying number of addresses. According to address reference there are three address, two address, one address and zero address reference instructions. Let us see examples of each of them.

Three address instructions : The three address instruction can be represented symbolically as

```
ADD A, B, C
```

where A, B, C are the variables. These variable names are assigned to distinct locations in the memory. In this instruction operands A and B are called source operands and operand C is called destination operand and ADD is the operation to be performed on the operands. Thus the general instruction format for three address instruction is

```
Operation Source 1, Source 2, Destination
```

The number of bits required to represent such instruction include :

1. Bits required to specify the three memory addresses of the three operands. If n-bits are required to specify one memory address, 3n bits are required to specify three memory addresses.
2. Bits required to specify the operation.

Two address instructions : The two address instruction can be represented symbolically as

```
ADD A, B
```

This instruction adds the contents of variables A and B and stores the sum in variable B destroying its previous contents. Here, operand A is source operand ; however operand B serves as both source and destination operand. The general instruction format for two address instruction is

```
Operation Source, Destination
```

To represent this instruction less number of bits are required as compared to three address instruction. The number of bits required to represent two address instruction include :

1. Bits required to specify the two memory addresses of the two operands, i.e. 2n bits.
2. Bits required to specify the operation.

One address instruction : The one address instruction can be represented symbolically as

```
ADD B
```

This instruction adds the contents of variable A into the processor register called **accumulator** and stores the sum back into the accumulator destroying the previous contents of the accumulator. In this instruction the second operand is assumed implicitly in a unique location accumulator. The general instruction format for one address instruction is

```
Operation Source
```

Few more examples of one address instructions are :

LOAD A : This instruction copies the contents of memory location A into the accumulator.

STORE B : This instruction copies the contents of accumulator into memory location B.

In one address instruction, it is important to note that the operand specified in the instruction can be either source operand or destination operand depending on the instruction. For example, in LOAD A instruction, the operand specified in the instruction is a source operand whereas the operand specified in the STORE B instruction is a

destination operand. Similarly, in one address instruction the implied operand (accumulator) can be either source or destination depending on the instruction.

Zero address instructions : In these instructions, the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a **pushdown stack**.

From above discussion we can easily understand that the instruction with only one address will require less number of bits to represent it, and instruction with three addresses will require more number of bits to represent it. Therefore, to access entire instruction from the memory, the instruction with three addresses requires more memory accesses while instruction with one address requires less memory accesses. The speed of instruction execution is mainly depend on how much memory accesses it requires for the execution. If memory accesses are more, more time is required to execute the instruction. Therefore, the execution time for three address instructions is more than the execution time for one address instructions.

To have a less execution time we have to use instructions with minimum memory accesses. For this instead of referring the operands from memory it is advised to refer operands from processor registers. When machine level language programs are generated by compilers from high-level languages, the intelligent compilers see that the maximum references to the operands lie in the processor registers.

Example 1.11.1 *Write a program to evaluate the arithmetic statement* $Y = (A + B) * (C + D)$ *using three-address, two-address, one-address and zero-address instructions.*

Solution :

Using three address instructions

```
ADD  A, B,R1     ;  R1 ← M[A] + M[B]
ADD  C, D,R2     ;  R2 ← M[C] + M[D]
MUL R1, R2,Y     ;  M[Y] ← R1 * R2
```

Using two address instructions

```
MOV  A, R1       ;  R1 ← M[A]
ADD  B, R1       ;  R1 ← R1 + M[B]
MOV  C, R2       ;  R2 ← M[C]
ADD  D, R2       ;  R2 ← R2 + M[D]
MUL  R2, R1      ;  R1 ← R1 * R2
MOV R1, Y        ;  M[Y] ← R1
```

Using one address instruction

```
LOAD    A        ;  AC ← M[A]
ADD     B        ;  AC ← AC + M[B]
STORE   T        ;  M[T] ← AC
LOAD    C        ;  AC ← M[C]
```

```
ADD     D       ;   AC ← AC + M[D]
MUL     T       ;   AC ← AC * M[T]
STORE   Y       ;   M[Y] ← AC
```

Using zero address instructions

```
PUSH    A       ;   TOS ← A
PUSH    B       ;   TOS ← B
ADD             ;   TOS ← (A + B)
PUSH    C       ;   TOS ← C
PUSH    D       ;   TOS ← D
ADD             ;   TOS ← (C + D)
MUL             ;   TOS ← (C + D) * (A + B)
POP     Y       ;   M[Y] ← TOS
```

Example 1.11.2 *Write the program to evaluate the expression* $X = \frac{A\,[B+C(D+E)]}{F(G+H)}$ *using the zero address instructions and one address instructions.*

Solution :

Program using zero address instructions

```
PUSH    D       ;   TOS ← D
PUSH    E       ;   TOS ← E
ADD             ;   TOS ← (D + E)
PUSH    C       ;   TOS ← C
MUL             ;   TOS ← C × (D + E)
PUSH    B       ;   TOS ← B
ADD             ;   TOS ← B + C × (D + E)
PUSH    A       ;   TOS ← A
MUL             ;   TOS ← A [B + C × (D + E)]
PUSH    G       ;   TOS ← G
PUSH    H       ;   TOS ← H
ADD             ;   TOS ← G + H
PUSH    F       ;   TOS ← F
MUL             ;   TOS ← F × (G + H)
DIV             ;   TOS ← A[B + C × (D + E)]/F × (G + H)
POP     X       ;   M [X] ← TOS
```

Program using one address instructions

```
LOAD    H       ;   AC ← M[H]
ADD     G       ;   AC ← AC + M[G]
MUL     F       ;   AC ← AC * M[F]
STORE   T       ;   M[T] ← AC
LOAD    D       ;   AC ← M[D]
```

```
ADD     E    ;   AC ← AC + M[E]
MUL     C    ;   AC ← AC * M[C]
ADD     B    ;   AC ← AC + M[B]
MUL     A    ;   AC ← AC * M[A]
DIV     T    ;   AC ← AC/M[T]
STORE   X    ;   M[X] ← AC
```

Example 1.11.3 $X = A \times B + C \times C$

Explain how the above expression will be executed in one address, two address and three address processors in an accumulator organization. **Dec.-03, 10, Marks 8**

Solution :

Using one address instruction

```
LOAD     A    ;    AC ← M[A]
MUL      B    ;    AC ← AC * M[B]
STORE    T    ;    M[T] ← AC
LOAD     C    ;    AC ← M[C]
MUL      C    ;    AC ← AC * M[C]
ADD      T    ;    AC ← AC + M[T]
STORE    X    ;    M[X] ← AC
```

Using two address instructions

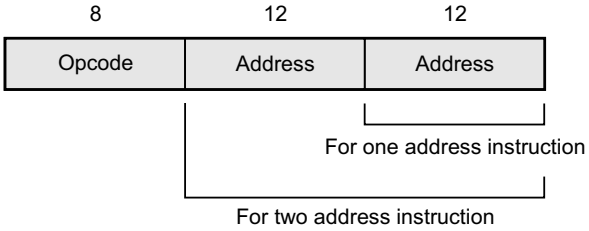
```
MOV  A, R1      ;    R1 ← M[A]
MUL  B, R1      ;    R1 ← R1 * M[B]
MOV  C, R2      ;    R2 ← M[C]
MUL  C, R2      ;    R2 ← R2 * M[C]
ADD R2, R1      ;    R1 ← R1 + R2
MOV  R1, X      ;    M[T] ← R1
```

Using three address instructions

```
MUL  A, B, R1    ;    R1 ← M[A] * M[B]
MUL  C, C, R2    ;    R2 ← M[C] + M[C]
add   R1, R1, X  ;    M[X] ←R1 * R2
```

Example 1.11.4 *A computer has 32-bit instructions and 12-bit addresses. If there are 250 two-address instructions, how many one address instructions can be formulated ?*

Solution : The given instruction is 32-bit. These 32-bit has following format.



The total number of instruction combinations for two address instructions are :

$$2^8 = 256$$

In these 256 combinations, 250 combinations are used for two address instructions. Therefore, remaining 6 combinations are used for one address instructions. Thus, the maximum number of one address instructions :

$$= (256 - 250) \times 2^{12} = 6 \times 12^{12}$$

$$= \textbf{24576 instructions}$$

Example 1.11.5 *A machine support 16-bit instruction format, the size of address field is 4-bits. The computer support expanding opcode technique and has 55000 zero-address instruction and 100 one address instructions. Find out the number of two address instructions supported by this machine.*

Solution : A 16-bit instruction can supports 2^{16} = 65536 zero address instruction. It is given that machine supports 55000 zero address instructions. Thus we have 65536 – 5500 = 10536 instructions combinations to in corporate one address and two address instructions. A 100 one address instructions requires 100×2^4 (four bit address field) = 1600 instruction combinations. Thus 10536 – 1600 = 8936 instruction combinations left for two-address instructions. Let us consider the number of two address instructions supported by this machine is x then

$$8936 > x \times 2^8 \text{ (eight bit address field)}$$

$$\therefore \quad x = 34.9 \approx \textbf{34 instructions}$$

So there are 34 two-address instructions supported by this machine.

Review Questions

1. *What do you mean by instruction format ? Explain it with the help of example.*
2. *Give an example each of zero address, one-addressing two address and there address instructions.* **May-06, Marks 2**
3. *Give the different instruction formats of a CPU in general.* **Dec.-06, Marks 6**
4. *Write a note on instruction formats.* **Dec.-07, May-08, Marks 4**
5. *Explain zero, one, two, and three addressing instruction with example.* **May-09, Marks 8**
6. *What are the different types of addressing instructions ? Give examples to each.* **Dec.-10, Marks 2**

1.12 Instruction Types

May-06, 12 Dec.-06, 12

A computer has a set of instructions that allows the user to formulate any data-processing task. To carry out tasks, regardless of whether a computer has 100 instructions or 300 instructions, its instructions must be capable of performing following basic operations :

- Data movement
- Data processing
- Program sequencing and control.

1.12.1 Data Transfer Instructions

Load Instruction : This instruction copies contents of memory location into the accumulator or general purpose register.

Examples

1. Load M : Copies the contents of memory location M into the accumulator.
2. Load M, Ri : Copies the contents of memory location M into Ri (Ri represents general purpose register)

Store Instruction : This instruction copies the contents of accumulator or general purpose register into memory location.

Examples

1. Store M : Copies the contents of accumulator into memory location M.
2. Store Ri, M : Copies the contents of Ri into memory location M.

Move Instruction : This instruction places a copy of the contents of source into destination.

Examples

```
1.   MOVE    A, Ri  ;    same as Load A, Ri
2.   MOVE    Ri, A  ;    same as store Ri, A
```

1.12.2 Arithmetic Instructions

Add Instruction : This instruction adds the contents of memory location or register (source) and destination, and store result in the destination.

Examples

1. Add M : Add the contents of memory location M to the contents of the accumulator and place the sum back into the accumulator
2. Add Ri, Rj : Add the contents of Ri to the contents of Rj and store the sum into Rj
3. Add Ri, Rj, Rk : Add the contents of Ri, Rj and Rk, and store the sum into Rk.
4. Add #50, R1 ; Add 50 into the contents of R1 and store and sum in R1.

Increment Instruction : This instruction increments the contents of specified register by 1.

Examples :

Increment R1 : R1 ← R1 + 1

Decrement Instruction : This instruction decrements the contents of specified register by 1.

Example

Decrement R2 : R2 ← R2 – 1

Sub Instruction : This instruction subtracts the contents of memory location or register (source) from destination and store result into destination.

Examples :

1. Sub M ; Subtracts the contents of memory location M from the contents of accumulator and store the result back into the accumulator.
2. Sub Ri, Rj : Subtract the contents of Ri from Rj and store the result in Rj

Compare Instruction : Subtracts the contents of memory location or register (source) from destination and sets the condition code flags according to the result. It does not change the value of either operand.

Examples

1. Compare M : Compares contents of memory location M with the contents of accumulator
2. Compare #50, R1 : Compares 50 with the contents of R1.

Clear Instruction : This instruction clears (makes contents equal to 0) specified register.

Example

Clear R1 : R0 ← 0

Multiplication and Division Instructions

The multiplication instruction does the multiplication of two operands and stores the result in the destination operand. On the other hand the division instruction does the division of two operands and stores the result in the destination operand. For example

```
MUL  R1, R0      ; R1 ← R1 × R0
DIV  R1, R0      ; R1 ← R1 ÷ R0
```

Unfortunately, all processors do not provide these instructions. On those processors MUL and DIV instruction are implemented by performing basic operations such as add, subtract, shift and rotate, repeatedly.

1.12.3 Logical Instructions

Logic instructions perform logic operations such as AND, OR and NOT between registers or between register and memory location. In these instructions the logic operation is applied to individual bits of the instruction. Let us see various examples.

1. NOT dst :

This instruction complements all bits contained in the destination operand, changing all 0s to 1s and all 1s to 0s.

This instruction is used to find the 1's complement and 2's complement of a given number. The following instruction sequence gives 1's complement and then 2's complement of a number in R1 register.

```
NOT R1          ; 1's complement
ADD R1,#1       ; 2's complement
```

Many processors support the instruction called Negate which does the task of above two instructions.

```
Negate R1
```

This instruction gives 2's complement of a number in R1 register.

2. AND R0, #% 10001001 : This instruction logically ANDs the contents of register R0 and the given number in binary form. If the contents of R0 are 1111 1010, then after ANDing these contents bit by bit with the given binary number we get result as 1000 1000.

```
1111 1010
1000 1001
---------
1000 1000
```

3. OR R0, # $ 3E : This instruction logically ORs the contents of register R0 and the given number in hexadecimal form. If the contents of R0 are 1010 0010, then after ORing these contents bit by bit with the given number we get result as 1011 1110.

```
1010 0010
0011 1110
---------
1011 1110
```

1.12.4 Shift and Rotate Instructions

This instruction shift or rotates the bits of the operand right or left by some specified number of bit positions.

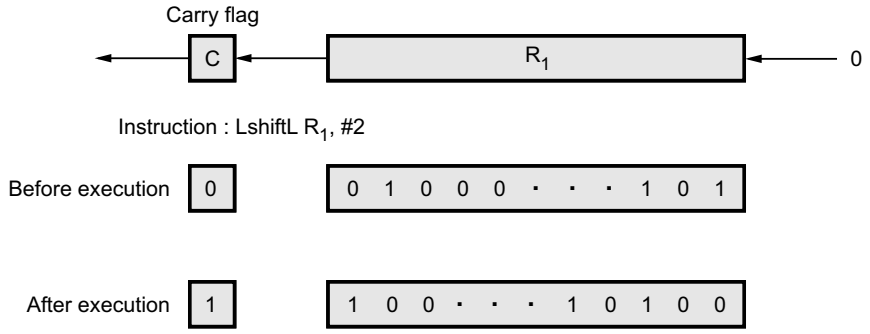
Logical Shifts

There are two logical shift instructions logical shift left (LShiftL) and logical shift right (LShiftR). These two instructions shift an operand by a number of the positions specified in a count operand contain in the instruction. The general syntax for these instructions are :

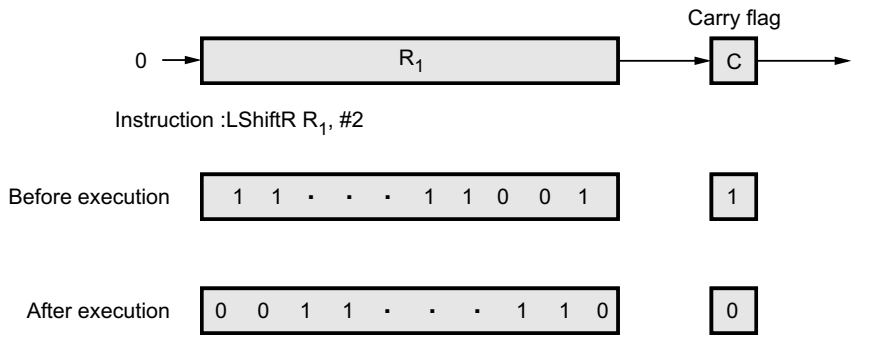
LshiftL dst, count

LshiftR dst, count

The count operand may be an immediate number, or it may be contains of the processor register. The Fig. 1.12.1 shows the operation of LShiftL and LShiftR instructions.



(a) Example of LshiftL instruction



(b) Example of LshiftR instruction

Fig. 1.12.1

Arithmetic Shift

In logical shift operation we have seen that the vacant positions created within the register due to shift operation are filled with zeroes. In arithmetic right shift it is necessary to repeat the sign bit as the fill-in bit for the vacated position. This requirement on right shifting distinguishes arithmetic shifts from logical shifts. Otherwise two shift operations are very similar. The Fig. 1.12.2 shows the example of an arithmetic right shift (AshiftR). The arithmetic left shift (AshiftL) is same as the logical left shift.

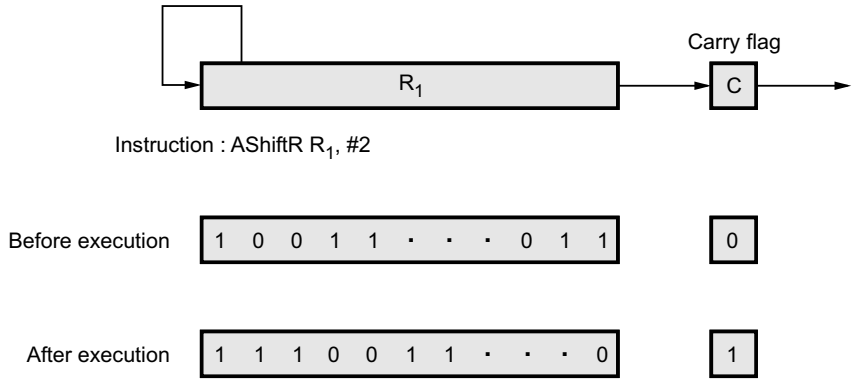


Fig. 1.12.2 Example of AshiftR instruction

Rotate Operations

In shift instructions, the bits shifted out of the operand are lost, except for last bit shifted out which is retained in the carry flag C. The rotate instructions, on the other hand, preserves all bits. They move the bits that are shifted out of one end of the operand back into the other end. The basic two types of rotate instructions are rotate left and rotate right. These two instructions are further subdivided into the rotation with carry and rotation without carry. The Fig. 1.12.3 shows the rotate instructions with and without carry flag being included in the rotation. When carry flag is not included in the rotation it retains the last bit of rotation.

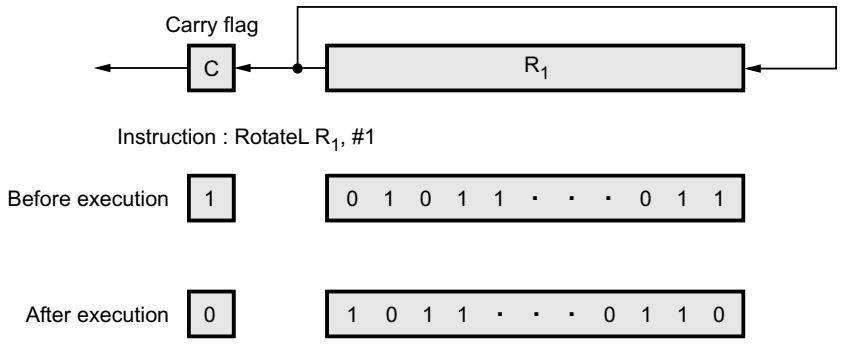


Fig. 1.12.3 (a) Example of RotateL instruction

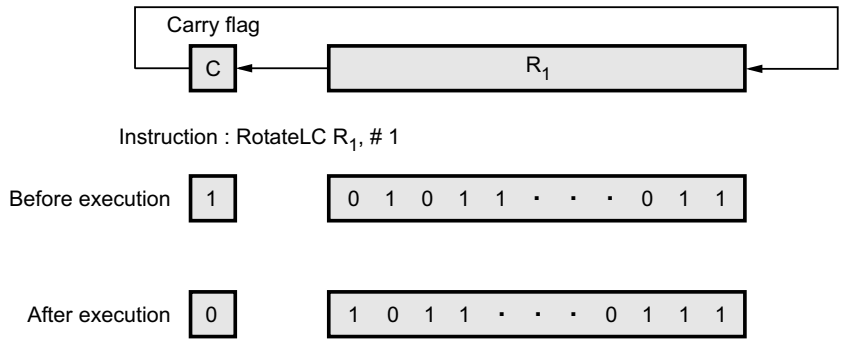


Fig. 1.12.3 (b) Example of RotateLC instruction

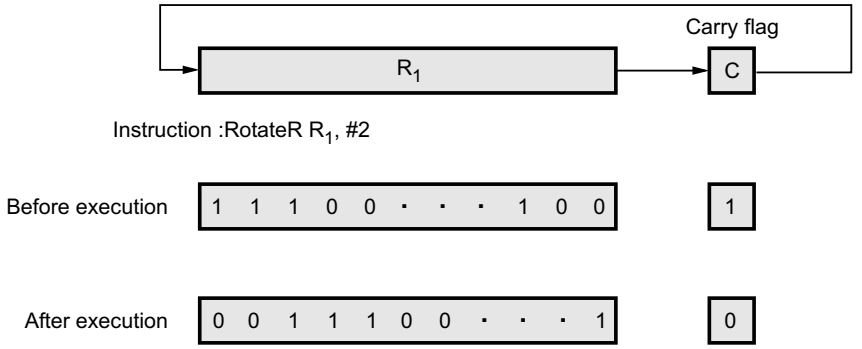


Fig. 1.12.3 (c) Example of RotateR instruction

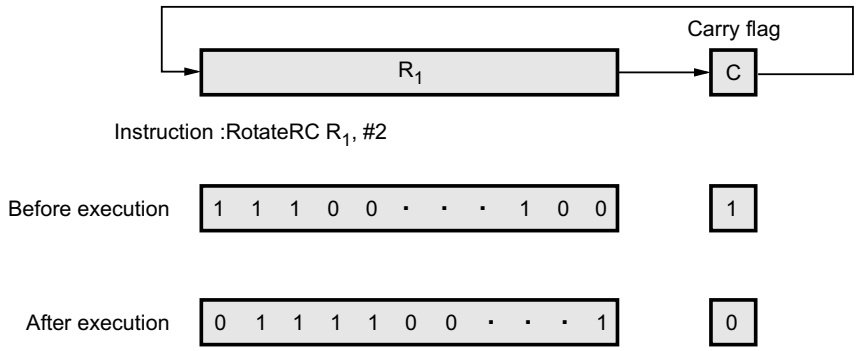


Fig. 1.12.3 (d) Example of RotateRC instruction

1.12.5 Program Sequencing and Control Instructions

Branch Instructions

There are two types of branch instructions : Unconditional branch and conditional branch. Unconditional branch instruction transfer the program flow to the target address. On the other hand conditional branch instruction checks the condition specified in the branch instruction and if the condition is satisfied it transfers the program flow to the target address; otherwise it does nothing and next instruction in the sequence is executed.

Example

Branch Loop : This instruction transfers the program flow to the location loop

BGTZ Loop : This instruction transfer the program flow to the location loop only if the result of previous operation is greater than zero.

Call and Return Instructions

Before going to see call and return instructions let us know about subroutines. A subroutine is a group of instructions, that performs a particular subtask which is executed number of times. It is written separately. The processor executes this subroutine by transferring program control to the subroutine program. After completion of subroutine program execution, the program control is returned back to the main program. The use of subroutines is a very important technique in designing software for processor systems because it eliminates the need to write a subtasks repeatedly; thus it uses memory more efficiently. When a program branches to a subroutine we say that it is **calling** the subroutine. The instruction that performs the branch operation is referred to as **Call** instruction. After completion of subroutine program execution, the program control is returned back to the main program by executing a **Return** instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location, i.e., the next instruction from where the subroutine is **called** from the calling program. When the call instruction is executed the PC is updated to point to the next instruction. The updated contents of the PC are saved by the call instruction to enable correct return to the calling program.

The technique by which a computer makes it possible to call and return from subroutines is referred to as its **subroutine linkage** method. The simplest method is to save the return address in a register dedicated to this function. Such a register is called **link register.** Let us see how Call and Return operations are performed.

Call Operation :

- It stores the contents of the PC in the link register.
- It then branches the program control to the target address specified by the instruction.

Return Operation :

- It loads the PC with the contents of link register and branches to the next instruction from where the subroutine is called.

The Fig. 1.12.4 illustrates the Call and Return operations.

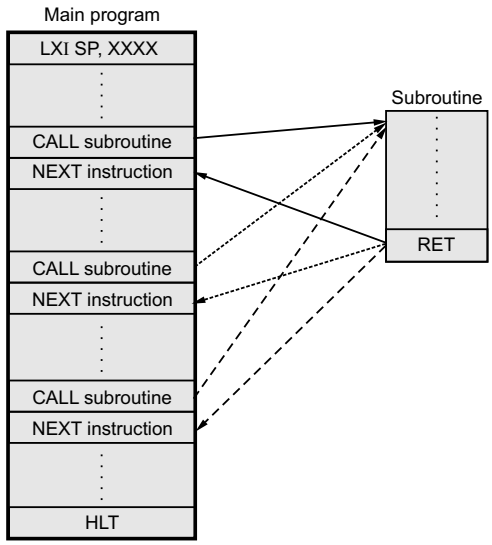


Fig. 1.12.4 (a) Program flow

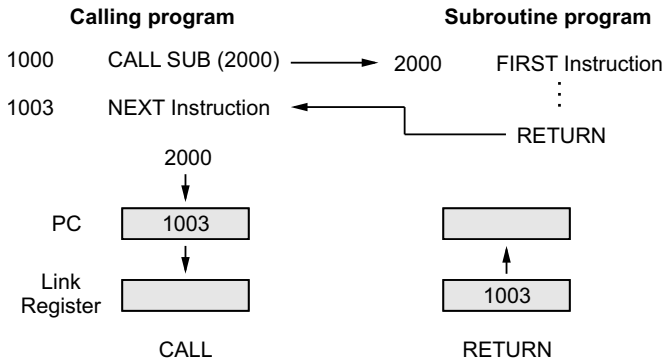


Fig. 1.12.4 (b) Subroutine linkage using a link register

1.12.6 RISC Instructions

RISC refers to Reduced Instruction Set Computers or Computing. RISC microprocessors are very different from CISC microprocessors. RISC use concept of keeping the instruction set as simple as possible to allow the microprocessor's program to be written using only simple instructions.

The features of RISC Instructions are

- In RISC processors, there is an **one instruction per machine cycle**.
- RISC machine **instructions are not complicated** and can execute about as fast as, microinstruction on CISC machines.
- The machine **instructions are hardwired**. These instructions are executed faster than the instructions implemented with microinstructions.
- RISC processor provides **limited number of instructions,** which simplifies the design of control unit.
- RISC processor uses **simple addressing modes**. Almost all instructions use simple register addressing.
- RISC processors use **simple instruction formats with fixed instruction length**. The instruction length is aligned on word boundaries. Field locations, especially opcodes are fixed.

Review Questions

1. *Explain the various instruction types.* **Dec.-12**
2. *Explain in detail the different types of instructions that are supported in a typical processor.* **May-06, Marks 10**
3. *What are the four basic types of operations that need to be supported by an instruction set ?* **Dec.-06, Marks 2**
4. *Describe the different types of instructions with an example for each.* **May-12, Marks 8**

1.13 Addressing Modes

Dec.-06, 07, 08, 10, 11, 12, May-04, 06, 07, 08, 09, 10, 11, 12

Part of the programming flexibility for each processor is the number and different kind of ways the programmer can refer to data stored in the memory or I/O device. The different ways that a processor can access data are referred to as **addressing schemes** or **addressing modes.**

1.13.1 Implementation of Variables and Constants

The variables and constants are the most commonly used data types on every computer. In assembly language, a variable is initialized by assigning register or a memory location to hold its value. Thus by changing the contents of specified register or memory location it is possible to change the value of the variable. Let us see the addressing modes used for this purpose.

1. Register Mode : The operand is the contents of processor register. The name of register is specified in the instruction.

Example : MOV R1, R2

This instruction copies the contents of register R2 to register R1.

2. Absolute Mode or Direct Mode : The address of the location of the operand is given explicitly as a part of the instruction.

Example : MOVE A, 2000

The above instruction copies the contents of memory location 2000 into the A register. As shown in the instruction, here, address of operand is given explicitly in the instruction.

The constant for address and data can be represented by immediate addressing mode in the assembly language programming. Let us see immediate addressing mode.

3. Immediate Mode : The operand is given explicitly in the instruction.

Example : MOVE A, #20

The above instruction copies operand 20 in the register A. The sign # in front of the value of an operand is used to indicate that this value is an immediate operand.

1.13.2 Indirection and Pointers

Some processor instructions does not provide an operand or its address explicitly. Instead, they provide the information from which the memory address of the operand can be determined. Such address is referred to as **effective address** (EA) of the operand.

4. Indirect Mode : The effective address of the operand is the contents of a register or the main memory location whose address is given explicitly in the instruction. When the effective address of the operand is the contents of a register this addressing is known as **register indirect addressing**.

Example : MOVE A, (R0)

The above instruction copies the contents of memory addressed by the contents of register R0 into the register A.

The indirection can be denoted by placing the name of register or the memory address given in the instruction in parentheses. For example : MOVE (LOC), R1. This instruction copies the contents of register R1 into the memory location whose address is stored at the memory location LOC.

The register or memory location which provides the address of operand is known as **pointer**.

1.13.3 Indexing and Arrays

The indexing is a technique that allows programmer to point or refer the data (operand) stored in sequential memory locations one by one.

5. Index Mode : The effective address of the operand is generated by adding a constant value (specified in the instruction) to the contents of a register.

Example : MOVE 20 (R1), R2

The above instruction loads the contents of register R2 into the memory location whose address is calculated by addition of the contents of register R1 and constant value (offset or displacement) 20.

Arrays are the most common way to structure the data. The array data structure is easy to handle and moreover it is supported by almost all the programming languages such as 'C', Pascal, Fortran etc. Many programming problems can be solved using array.

For example : We need the information about the marks obtained by the students in a particular subject or we want to know salary of every employee in some company. Then such information can be collectively stored in array data structure.

We can visualize an array as :

0	10
1	20
2	40
3	70
4	101
5	30
6	45
7	78
8	75
9	100

Here a [2] = 40
a [8] = 75
and so on

Fig. 1.13.1 Array a[10]

Note that the elements in array are always stored in contiguous memory locations. These array elements can be accessed by using indirect or index addressing modes.

The two dimensional array is something which you can compare with the two storied building! Extra space which is arranged in rows and columns. Let me draw a figure which will represent the two dimensional array.

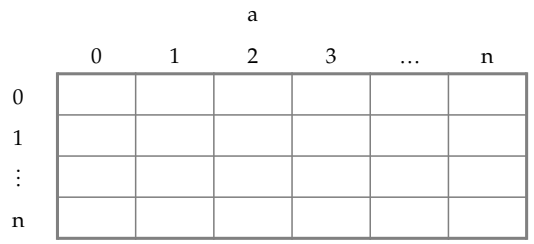


Fig. 1.13.2 2 Dimensional array

The two dimensional array elements can be accessed by index addressing mode. Using index addressing mode we can specify the row address in the register and offset gives the column address. For example : MOV R2, 10(R1). In this instruction R1 specifies the row the address and offset 10 gives the column address.

1.13.4 Relative Addressing

We have seen the index addressing mode using general-purpose processor registers. If we use this addressing mode with program counter instead of the general purpose register we get the relative addressing mode.

6. Relative Mode : The effective address is determined by the index mode using program counter in place of the general purpose processor register.

This addressing mode is commonly used to specify the target address in branch instructions. For example,

JNZ BACK

This instruction causes program execution to go to the branch target location identified by the name BACK, if the branch condition is satisfied. The branch target location can be determined by specifying it an offset from the current value of the program counter. Since the branch target location may either before or after the branch instruction, the offset is specified as a signed number.

1.13.5 Additional Modes

So far we have discussed the six addressing modes : Register, absolute (direct), immediate, indirect, index and relative addressing modes. In this section we will see two more addressing modes : autoincrement and autodecrement.

7. Autoincrement Mode : The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are incremented to address the next location.

Example : MOVE (R2), + R0

The above instruction copies the contents of register R0 into the memory location whose address is specified by the contents of register R2. After copy operation, the contents of register R2 are automatically incremented by 1.

8. Autodecrement Mode : The contents of a register specified in the instruction are decremented and then they are used as an effective address to access a memory location.

Example : MOVE R1, – (R0)

This instruction, initially decrements the contents of register R0 and then the decremented contents of register R0 are used to address the memory location. Finally, the contents from the addressed memory location are copied into the register R1.

Example 1.13.1 *Registers R1 and R2 of a computer contain the decimal values 1200 and 2400 respectively. What is the effective address of the memory operand in each of the following instructions?*

1) Load 20 (R1), R5 2) Add - (R2), R5

3) Move # 3000, R5 4) Sub (R1) +, R5 **May-06, Marks 6**

Solution : R1 1200 R2 2400

Sr. No.	Instruction	Effective address of memory
1	Load 20(R1), R5	1200 + 20 = 1220
2	Add – (R2), R5	2400 – 1 = 2399
3	Move #3000, R5	3000
4	Sub (R1)+, R5	1200

Example 1.13.2 *Registers R1 and R2 of a computer contain the decimal values 1200 and 4600. What is the effective address of the memory operand in each of the following instructions ?*

a) Load 20 (R1), R5 b) Add– (R2), R5. **May-08, Marks 2**

Solution : a) EA : 1200 + 20 b) EA : 4599.

Example 1.13.3 *What must the address field of an indexed addressing mode instruction be to make it the same as a register indirect mode instruction ?*

Solution : The effective address in the indexed mode is given by EA = Offset + R and the effective address in the register indirect mode is given by EA = R. Thus, when offset is 0, the indexed addressing mode instruction is same as that of register indirect mode instruction.

Example 1.13.4 *At memory address 200, two word instruction, load to AC is stored with a mode bit as a most significant bit. At location 201 the address stored is 500. At location 202 next instruction is stored. The following numbers are stored at different memory locations as shown ahead.*

Memory location (Address)	Memory contents
399	450
400	700
500	800
600	900
702	325
800	300

If the content of PC is 200, while the contents of register R1 is 400. XR register is 100. If all the numbers and addresses are in decimal number, find out contents of AC and effective address for the following addressing modes.

i) Direct address ii) Indirect address

iii) Relative address iv) Indexed address

v) Register indirect addressing mode.

Solution : The Fig. 1.13.3 shows the given information for our convenience.

i) Direct address : EA = 500 ∴ AC = **800**

ii) Indirect address : EA = [500] = **800** ∴ AC = **300**

iii) Relative address : EA = PC + Address part of instruction
= 202 + 500 = **702**
∴ AC = **325**

iv) Indexed address : EA = XR + Address part of instruction
= 100 + 500 = **600**
∴ AC = **900**

v) Register indirect address : EA = R1 = **400**
∴ AC = **700**

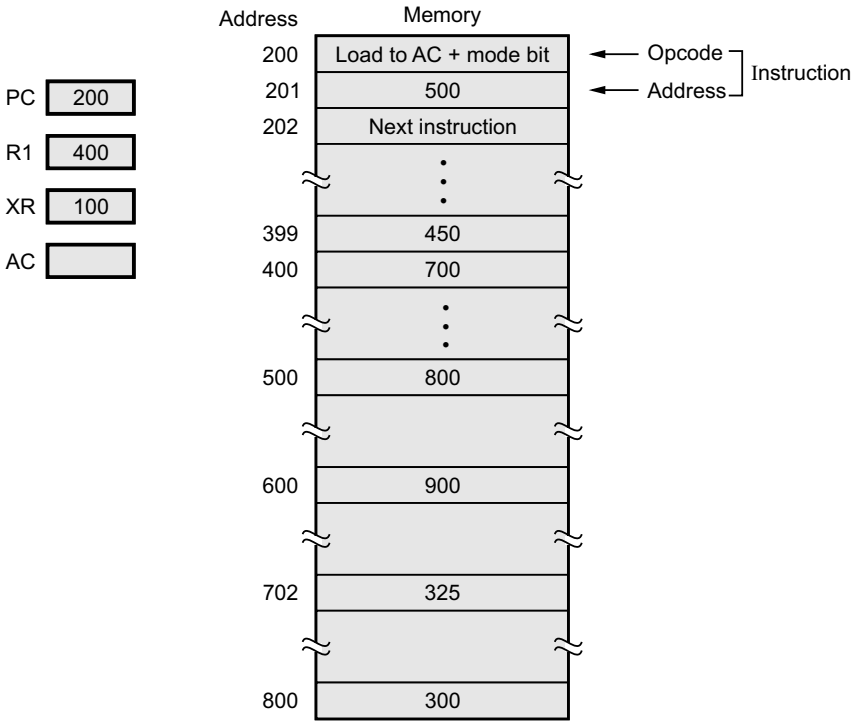


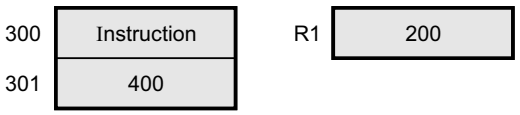
Fig. 1.13.3

Example 1.13.5 *An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is :*

i) Immediate ii) Direct iii) Register indirect

iv) Relative v) Index with R1 as the index register.

Solution :



i) Immediate mode : In immediate addressing mode, the address bits of an instruction code specify the actual operand. Hence for this addressing mode the address of the operand i.e. effective address is 301.

ii) Direct mode : Effective Address (EA) : **400**

iii) Register indirect node : Effective Address (EA) : **200**

iv) Relative mode : Effective Address (EA) : PC + Address part of instruction = 302 + 400 = **702**

v) Index with R1 as the index register : Effective address (EA)

= R1 + Address part of the instruction = 200 + 400

= **600**

Review Questions

1. *Write a short note on addressing modes.*
2. *Define addressing mode. Classify addressing modes and explain each type with examples.* **Dec.-06, May-12, Marks 10**
3. *Describe different types of addressing modes in detail.* **May-07, Marks 8**
4. *What is the information conveyed by addressing modes?* **Dec.-07, Marks 4**
5. *List the various addressing modes. Give a brief explanation of each of them with an example.* **Dec.-07, Marks 8**
6. *Describe various addressing modes with suitable examples.* **Dec.-07, Marks 16**
7. *Explain the different types of addressing modes with suitable examples.* **May-08, Marks 16**
8. *Differentiate direct and indirect addressing mode.* **Dec.-08, Marks 2**
9. *Describe the addressing mode for accessing memory content.* **May-09, Marks 8**
10. *Explain briefly the various types of addressing modes with example.* **May-09, Marks 16**
11. *What is the need for different addressing modes? Explain the various addressing modes with examples.* **May-10, Marks 16**
12. *Define addressing mode. Classify addressing modes and explain each type with example.* **Dec.-10, Marks 16**
13. *Describe in detail the different kinds of addressing mode with an example.* **Dec.-11, 12, Marks 8**

Two Marks Questions with Answers

Q.1 ***Define computer architecture.***

Ans. : Computer architecture is defined as the functional operation of the individual H/W unit in a computer system and the flow of information among the control of those units.

Q.2 ***Define computer H/W.***

Ans. : Computer H/W is the electronic circuit and electro mechanical equipment that constitutes the computer.

Q.3 *Name the functional units of a computer.* **Dec.-08**

Ans. : The functional units of a computer are :

1. Input unit 2. Output unit 3. Control unit
4. Memory unit 5. Arithmetic and logical unit

Q.4 *What is meant by Central Processing Unit (CPU) ?* **Dec.-09**

Ans. : The arithmetic and logic unit in conjunction with control unit is commonly called **Central Processing Unit (CPU)**.

Q.5 *What is the function of input unit ?*

Ans. : A computer accepts a digitally coded information through input unit using input devices such as keyboard and mouse.

Q.6 *What are the functions of control unit ?*

Ans. : The control unit co-ordinates and controls the activities amongst the functional units. The basic function of control unit is to fetch the instructions stored in the main memory, identify the operations and the devices involved in it, and accordingly generate control signals to execute the desired operations.

Q.7 *What is the function of arithmetic and logic unit ?*

Ans. : The arithmetic and logic unit (ALU) is responsible for performing arithmetic operations such as add, subtract, division and multiplication, and logical operations such as ANDing, ORing, Inverting etc.

Q.8 *What is meant by the stored program concept ? Discuss.* **May-07**

Ans. : According to stored-program concept memory can contain the program (source code), the corresponding compiled machine code, editor program and even the compiler that generated the machine code.

Q.9 *What are the main features of a Von Neumann model ?*

Ans. : The main features of a Von Neumann model are :

1. It uses stored program concept. The program (instructions) and data are stored in a single read-write memory.
2. The contents of read-write memory are addressable by location, without regard to the type of data contained there.
3. Execution of instructions occurs in a sequential manner (unless explicitly modified) from one instruction to the next.

Q.10 *What do you mean by Von Neumann bottleneck ?* **Dec.-12**

Ans. : Because of the stored program architecture of Von-Neumann machine, the processor performance is tightly bound to the memory performance. That is, since we need to access memory at least once per cycle to read an instruction, the processor can

only operate as fast as the memory. This is sometimes known as the **Von Neumann bottleneck** or **memory wall**.

Q.11 *What is the role of MAR and MDR ?*

Ans. : The MAR (memory address register) is used to hold the address of the location to or from which data are to be transferred and the MDR(memory data register) contains the data to be written into or read out of the addressed location.

Q.12 *What is the role of IR ?*

Ans. : Instruction Register (IR) contains the instruction being executed. Its output is available to the control circuits, which generate the timing signals for controlling the processing circuits needed to execute the instructions.

Q.13 *What is the role of PC ?*

Ans. : The Program Counter (PC) register keeps track of the execution of the program. It contains the memory address of the instruction currently being executed . During the execution of the current instruction, the contents of the PC are updated to correspond to the address of the next instructions to be executed.

Q.14 *Define unsolvable problem.*

Ans. : Refer section 1.1.3.

Q.15 *Define undecidable problem.*

Ans. : Refer section 1.1.3.

Q.16 *Define tractable and intractable problems.*

Ans. : Refer section 1.1.3.

Q.17 *Define CPI.*

Ans. : The term clock cycles per instruction is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI.

CPI= CPU clock cycles/Instruction count.

Q.18 *Define MIPS.*

Ans. : MIPS is a way to measure the processor speed. The processor speed can be measured in terms of **million instructions per second** (MIPS). It is given as

$$\text{MIPS rate} = \frac{1}{\text{Average time required for the execution of instruction} \times 10^6}$$

Q.19 *Define MIPS rate.*

Ans. : The rate at which the instructions are executed at a given time. It is given by

$$\text{MIPS rate} = \frac{N \times R}{C \times 10^6}$$

Q.20 *Define throughput and throughput rate.*

Ans. : Throughput is defined as the total amount of work done in a given time.

Throughput rate is defined as the rate at which the total amount of work done at a given time.

Q.21 *Define SPEC rating.*

Ans. : The SPEC rating is given by

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

Q.22 *State and explain the performance equation.* **Dec.-12**

Ans. : Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock cycle rate is R cycles per second, the program execution time is given by

$$T = (N \times S) / R$$

This is often referred to as the basic performance equation.

Q.23 *Define clock rate.*

Ans. : The clock rate is given by,

R = 1/P, where P is the length of one clock cycle.

Q.24 *Define response time and throughput.*

Ans. : Response time is the time between the start and the completion of the event. Also referred to as execution time or latency. Throughput is the total amount of work done in a given amount of time

Q.25 ***Suppose that we are considering an enhancement to the processor of a server system used for web serving. The new CPU is 10 times faster on computation in the web serving application than the original processor. Assuming that the original CPU is busy with computation 40% of the time and is waiting for I/O 60% of the time. What is the overall speedup gained by incorporating the enhancement?***

Ans. : $\text{Frraction}_{\text{enhanced}} = 0.4$

$\text{Speedup}_{\text{enhanced}} = 10$

$\text{Speedup}_{\text{overall}} = 1/(0.6+0.4/10) = 1/0.64 = 1.56$

Q.26 ***What are the important factors that determine a computer's performance ? Give its significance.*** **Dec.-03**

Ans. : Refer section 1.3.4.

Q.27 *Give an example each of zero-address, one-address, two-address and three-address instructions.* **May-06**

Ans. : Refer section 1.11.

Q.28 *Which data structures can be best supported using (a) indirect addressing mode (b) indexed addressing mode?* **May-06**

Ans. : Indirect addressing mode supports pointers and indexed addressing mode supports arrays.

Q.29 *Explain in detail the different types of instructions that are supported in a typical processor.* **May-06**

Ans. :
- Data transfers between the main memory and the CPU registers
- Arithmetic and logic operation on data
- Program sequencing and control
- I/O transfers

Q.30 *What are the four basic types of operations that need to be supported by an instruction set ?* **Dec.-06**

Ans. : The following basic operations need to be supported by an instruction set :
- Data transfers between the memory and the processor registers.
- Arithmetic and logic operations on data.
- Program sequencing and control.
- I/O control.

Q.31 *What do you mean by zero address instruction ?*

Ans. : It is also possible to use instruction where the locations of all operand are defined implicitly. This operand of the use of the method for storing the operand in which called push down stack. Such instructions are sometimes referred to as zero address instructions.

Q.32 *What are addressing modes ?*

Ans. : The different ways that a processor can access data are referred to as **addressing schemes** or **addressing modes.**

Q.33 *List different addressing modes.*

Ans. :
1. Register Mode
2. Absolute Mode or Direct Mode
3. Immediate Mode
4. Indirect Mode
5. Index Mode
6. Relative Mode
7. Autoincrement Mode
8. Autodecrement Mode

Q.34 *Differentiate direct and indirect addressing mode.* **Dec.-08**

Ans. : In direct addressing mode the address of the location of the operand is given explicitly as a part of the instruction. On the other hand, in indirect addressing mode, the effective address of the operand is the contents of a register or the main memory location whose address is given explicitly in the instruction.

Q.35 *What are the steps in executing a program?*

Ans. : 1. Fetch 2. Decode
3. Execute 4. Store

Q.36 *What is a branch instruction?*

Ans. : Branch instruction is a type of instruction which loads a new value into the program counter.

Q.37 *What are the different types of address instructions? Give examples to each.* **Dec.-10**

Ans. : Refer section 1.11.

Q.38 *When addressing mode will be viewed critically ?* **Dec.-08**

Ans. : In case of pipelining addressing modes will be viewed critically.

Q.39 *Convert the following binary number to decimal number.*
a) 110101_2 *b)* 111100_2 **May-09**

Ans. :

$$(110101)_2 = 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$$
$$= 32 + 16 + 0 + 4 + 0 + 1 = (53)_{10}$$
$$(111100)_2 = 1\times 2^5 + 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 0\times 2^0$$
$$= 32 + 16 + 8 + 4 + 0 + 0 = (60)_{10}$$

Q.40 *How do you relate addition and subtraction ?*

Ans. : We can relate addition and subtraction operations of numbers by the following relationship :

$(\pm A) - (+B) = (\pm A) + (-B)$ and

$(\pm A) - (-B) = (\pm A) + (+B)$

Q.41 *What is one's complement of numbers ?*

Ans. : The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.

Q.42 *What is 2's complement of numbers ?*

Ans. : The 2's complement is the binary number that results when we add 1 to the 1's complement. It is given as 2's complement = 1's complement + 1.

Q.43 *Define IEEE floating point single and double precision standard.* **Dec.-06, 07**

Ans. : The 32-bit standard representation (**single-precision representation**) occupies a single 32-bit word. The 32-bits are divided into three fields as shown below :

(field 1) Sign ← 1 - bit

(field 2) Exponent ← 8 - bits

(field 3) Mantissa ← 23 - bits

The 64-bit standard representation (**double-precision representation)** occupies two 32-bit words. The 64-bits are divided into three fields as shown below :

(field 1) Sign ← 1 - bit

(field 2) Exponent ← 1 - bit

(field 3) Mantissa ← 52 - bits

Q.44 *In conforming to the IEEE standard mention any four situations under which a processor sets exception flag.* **Dec.-06**

Ans. : Underflow

Overflow

Divide by zero

Inexact

Invalid

Q.45 *Define underflow and overflow.* **May-08**

Ans. : Underflow : In a single precision, if the number requires an exponent less than – 126 or in a double precision, if the number requires an exponent less than – 1022 to represent its normalized form the underflow occurs.

Overflow : In a single precision, if the number requires an exponent greater than + 127 or in a double precision, if the number requires an exponent greater than + 1023 to represent its normalized form the overflow occurs.

Q.46 *How are the signed binary numbers represented?* **May-09**

Ans. : Refer section 1.9.1.

Q.47 *What are the various ways of representing signed integers in the system ?* **Dec.-07**

Ans. : Refer section 1.9.

Q.48 *How do you convert 2's complement to decimal?* **May-10**

Ans. : Step 1 : Convert decimal number to binary

Step 2 : Find 1's complement

Step 3 : Add 1.

Q.49 *What are the types of CPU organization?* **Dec.-10**

Ans. : 1. Accumulator based CPU organization
2. CPU with general register organization
3. CPU with pipelined architecture
4. CPU with CISC architecture
5. CPU with RISC architecture.

Q.50 *What do you mean by encoder ?*

Ans. : An encoder is a digital circuit that performs the inverse operation of a decoder. Encoder has 2^n (or fewer) input lines and n output lines. Encoder has enable inputs to activate encoded outputs.

Q.51 *What is decoder ?*

Ans. : A decoder is a multiple - input, multiple-output logic circuit which converts coded inputs into coded outputs, where the input and output codes are different. In a binary decoder n-inputs produce 2^n outputs. Usually, a decoder is provided with enable inputs to activate decoded output.

Q.52 *What is binary decoder ?*

Ans. : A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n outputs lines.

Q.53 *What is a data selector ? or what is multiplexer. or Why is MUX called as data detector.* **Dec.-06, May-11**

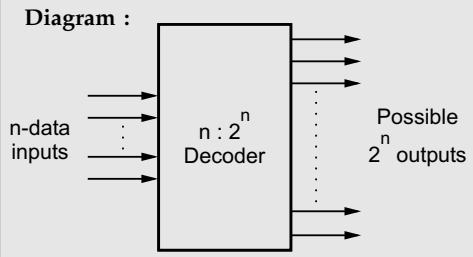
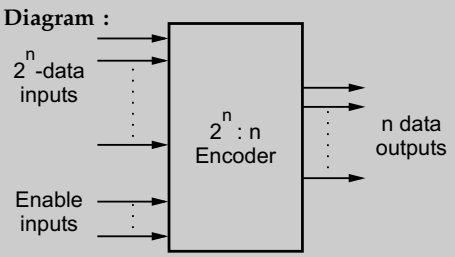
Ans. :

- Multiplexer is a digital switch. Particularly, it has 2^n input lines and n selection lines whose bit combinations determine which input line is selected and routed onto available only single output line.
- Hence, multiplexer is a selector of one out of several data sources available at its input lines, to connect it to output line. Simply it is a 'many into one' device and also called 'data selector'.

Q.54 *With a aid of block diagram clearly distinguish between a decoder and encoder.*

Ans. :

Sr. No.	Decoder	Encoder
1.	In decoder one of the output lines is activated corresponding to the binary input.	In encoder, the output lines generate the binary code, corresponding to the input value.

2.	Input of the decoder is an encoded information presented as n inputs producing 2^n possible outputs.	Input of the encoder is a decoded information presented as 2^n inputs producing n possible outputs.
3.	**Diagram :**  **General structure of decoder**	**Diagram :**  **General structure of encoder**
4.	The input code generally has a fewer bits than the output code	The input code generally has more bits than the output code.

Q.55 ***Distinguish between a demultiplexer and decoder.*** **Dec.-03, 05, May-04,**

Ans. :

Sr. No.	Decoder	Demultiplexer
1.	Decoder is a many inputs to many outputs device.	Demultiplexer is a one input to many outputs device.
2.	There are no selection lines.	The selection of specific output line is controlled by the value of selection lines.

Q.56 ***What are the applications of decoder ?*** **May-11**

Ans. : The applications of decoder are :

- Code converters
- Implementation of combinational circuits
- Address decoding
- BCD to 7-segment decoder

Q.57 ***List the speed enhancing features incorporated into the modern computers.***

Ans. : The speed enhancing features incorporated into the modern computers are :

- Parallel processing using multiple processing units.
- Instruction and arithmetic pipelining.
- Use of multiple-levels of cache memories.
- Use of VLSI technology.

Q.58 ***What are the factors to be considered in selecting a number representation to be used in a computer ?*** **Dec.-11**

Ans. :

- The number of types to be represented; for example, integers, real, float etc.
- The range of values likely to be encountered.

- The precision of the numbers, which refer to the maximum accuracy of the representation.
- The cost of the hardware required to store and process the numbers.

Q.59 *What is byte addressable memory?*

Ans. : The assignment of successive addresses to successive byte locations in the memory is called byte addressable memory.

Q.60 *What is little-endian?*

Ans. : When the lower byte addresses are used for the less significant bytes (the rightmost byte) of the word, addressing is called **little-endian.**

Q.61 *What are aligned addresses?*

Ans. : In general, words are said to be aligned in memory if they begin at the natural word boundaries. The addresses of such words are called **aligned addresses**.

Q.62 *What is big endian ?*

Ans. : When lower byte addresses are used for the most significant bytes (the leftmost bytes) of the word, addressing is called big endian.

Q.63 *State the drawbacks of relative addressing.*

Ans. : The main drawbacks of relative addressing are :

- need of extra logic circuits.
- Extra time required to compute addresses.

Q.64 *State the requirements to be satisfied by an instruction set.*

Ans. :

- It should be complete in the sense, i.e., it should be able to construct a machine language program to evaluate any function that is computable using a reasonable amount of memory space.
- It should be efficient, i.e., it should be able to performed frequently used functions rapidly with relatively few instructions.
- It should contain expected opcodes and sufficient addressing modes.
- To reduce hardware and software design cost it should be compatible with existing machines of the same computer family.

Q.65 *What is a register ?* **May-12**

Ans. : A group of flip-flops connected together forms a **register.** A register is used solely for storing and shifting data which is in the form of 1s and/or 0s, entered from an external source.

Q.66 *What is VLSI ?* **May-12**

Ans. : VLSI stands for very large scale integration. It allows manufacturers to fabricate a CPU, main memory or even all the electronic circuits of a computer, on a single IC.

❑❑❑

2 Data Path Design

Syllabus

Fixed point arithmetic, Addition, Subtraction, Multiplication and division, Combinational and sequential ALUs, Carry look ahead adder, Robertson algorithm Booth's algorithm, Non-restoring division algorithm, Floating point arithmetic, Coprocessor, Pipeline processing, Pipeline design, Modified Booth's algorithm.

Contents

2.1 Introduction

*2.2 Fixed Point Arithmetic . . . . . **Dec.-03, 08, May-08,** · · · · · · Marks 2*

*2.3 Adder and Subtractor Circuits **May-07, 08, 09, 10, 11,***
*. . . . . **Dec.-06 07, 08, 09,** · · · · · · Marks 16*

*2.4 Look Ahead Carry Adders **Dec.-06, 07, 08, 10,***
*. **May-06, 07, 08, 09,***
*. . . . . **10, 11, 12,** · · · · · · · · · · · · · · Marks 16*

*2.5 Multiplication . . . . . **Dec.-08, May-07,** · · · · · · · · Marks 16*

*2.6 Robertson Algorithm . . . . . **Dec.-03, 11,** · · · · · · · · · · · · Marks 16*

*2.7 Booth's Algorithm **Dec.-03, 08, 09, 10, 12***
*. . . . . **May-06, 07, 08, 09, 11** · · · · Marks 16*

*2.8 Fast Multiplication **May-06,07, 08, 09, 10, 11,***
*. . . . . **Dec.-07, 08,** · · · · · · · · · · · · Marks 8*

*2.9 Integer Division **Dec.-07, 08, 09,***
*. . . . . **May-06, 07, 09, 11, 12,** · · · Marks 16*

*2.10 Floating Point Arithmetic **Dec.-06, 07, 08, 12,***
*. . . . . **May-06, 09, 10,** · · · · · · · · · Marks 16*

*2.11 Arithmetic Logic Unit . . . . . **Dec.-07, 08,** · · · · · · · · · · · · Marks 16*

*2.12 Coprocessors . . . . . **Dec.-10, 11,** · · · · · · · · · · · · · Marks 2*

2.13 Pipeline Processing

*2.14 Pipeline Design **Dec.-03, 11,***
*. . . . . **May-06,** · · · · · · · · · · · · · · · Marks 16*

Two Marks Questions with Answers

2.1 Introduction

A processor consists of datapath (data processing) and control unit. The datapath unit performs all arithmetic and logical operations. We know that, modern computers communicate and operate with binary numbers which use only the digits 0 and 1. Based on binary number system various data type s are implemented in the computer system. The most common data types are : Fixed point numbers and Floating point numbers.

In this chapter, we see the implementation issues of fixed point and floating point arithmetic.

2.2 Fixed Point Arithmetic

Dec.-03, 08, May-08

2.2.1 Addition

Digital computers perform various arithmetic operations. The most basic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely,

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10_2$

The first three operations produce a sum whose length is one digit, but when the last operation is performed sum is two digits. The higher significant bit of this result is called a **carry** and lower significant bit is called **sum**. Such a operation is known as **half addition.** The operation which performs addition of three bits (two significant bits and a previous carry) is called a **full addition.**

The Table 2.2.1 shows the truth table for full addition.

Inputs			Outputs	
A	B	C_{in}	C_{out}	Sum (S)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 2.2.1 Truth table for full-adder

Addition method

1. Add bits column-wise starting from LSB with carry if any.
2. Put the sum at the bottom of the same column.
3. Put the carry, if any, on the top of next column.

Example 2.2.1 *Perform addition of $(11001100)_2$ and $(11011010)_2$.*

Solution :

	1	**1**		**1**	**1**				**Carry**
		1	1	0	0	1	1	0	0	**Number 1**
+		1	1	0	1	1	0	1	0	**Number 2**
	1	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**Result (S)**

In the above example, we have seen the addition of two unsigned binary numbers. In case of the signed numbers, we have to consider the sign of the number and we may require sign extension to avoid possible overflow.

Example 2.2.2 *Add $(28)_{10}$ and $(15)_{10}$ by converting them into binary.*

Solution : Using decimal to binary conversion technique we have,

$$(28)_{10} = (011100)_2 \text{ and } (15)_{10} = (01111)_2$$

		1	**1**	**1**			
Sign Extension → +	**0**	0	1	1	1	0	0
	0	**0**	0	1	1	1	1
Sign →	**0**	**1**	**0**	**1**	**0**	**1**	**1**

Carry
Binary equivalent of $(28)_{10}$
Binary equivalent of $(15)_{10}$
Result : Binary equivalent of $(43)_{10}$

$$\begin{array}{r} (28)_{10} \\ + \ (15)_{10} \\ \hline (43)_{10} \end{array}$$

2.2.2 Subtraction

We can relate addition and subtraction operations of numbers by the following relationship :

$(\pm A) - (+B) = (\pm A) + (-B)$ and

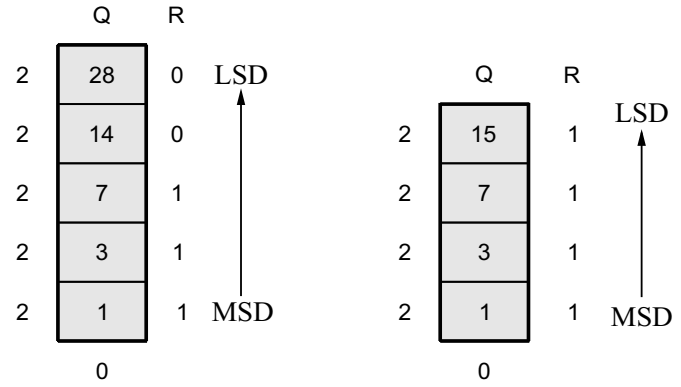
$(\pm A) - (-B) = (\pm A) + (+B)$

Therefore, we can change subtraction operation to an addition operation by changing the sign of the subtrahend.

Let us see the subtraction of binary number using 1's complement and 2's complement number representations.

Binary Arithmetic - Negative numbers in 1's complement form

Case 1 (Both Positive) : Add $(28)_{10}$ and $(15)_{10}$



$\therefore \quad (28)_{10} = (011100)_2$ $\therefore \quad (15)_{10} = (01111)_2$

Addition of 28 and 15 :

		1	**1**	**1**				**Carry**
	0	0	1	1	1	0	0	**$(28)_{10}$**
+	**0**	0	0	1	1	1	1	+ **$(15)_{10}$**
	0	**1**	**0**	**1**	**0**	**1**	**1**	**$(43)_{10}$**

Note : Here, the magnitude of greater number is 5-bit; however, the magnitude of the result is 6-bit. Therefore, the numbers are sign-extended to 7-bits.

Case 2 (Smaller Negative) : Add $(28)_{10}$ and $(-15)_{10}$

We have $(28)_{10} = (011100)_2$ and

$(15)_{10} = (01111)_2$

$\therefore$ 1's complement of 15 $= (10000)_2$

Addition of 28 and – 15 :

		1						**Carry**
		0	1	1	1	0	0	**$(28)_{10}$**
Sign extension	+	**1**	1	0	0	0	0	**$(-15)_{10}$**
	1	**0**	**0**	**1**	**1**	**0**	**0**	**Partial result**
							1	**Add end-around carry**
		0	0	1	1	0	1	**$(13)_{10}$**

Case 3 (Greater Negative) : Add $(-28)_{10}$ and $(15)_{10}$

We have $(28)_{10} = (011100)_2$ and

$(15)_{10} = (01111)_2$

$\therefore \quad (-28)_{10} = (100011)_2$

Addition of (– 28) and 15

		1	**1**	**1**	**1**		**Carry**
	1	0	0	0	1	1	**$(-28)_{10}$**
Sign extension +	**0**	0	1	1	1	1	**$(15)_{10}$**
	1	**1**	**0**	**0**	**1**	**0**	**$(-13)_{10}$**

Result is in 1's complement form

Verification :								
	1	1	0	0	1	0		Result
	0	0	1	1	0	1	→	complement of result $(13)_{10}$

Case 4 (Both Negative) : Add $(-28)_{10}$ and $(-15)_{10}$

We have $(28)_{10} = (011100)_2$ and

$(15)_{10} = (01111)_2$

$\therefore$ 1's complement of 15 $= (10000)_2$

1's complement of 28 $= (100011)_2$

Addition of (– 28) and (– 15) :

	1	**1**						
Sign-extension		**1**	1	0	0	0	1	1	**$(-28)_{10}$**
Sign-extension +	**1**	**1**	1	0	0	0	0	**$(-15)_{10}$**
Carry						**1**	**1**		**Carry**
	1	1	0	1	0	0	1	1	**Add end-around carry**
								1	**(–43) Result is in 1's complement form**
		1	0	1	0	1	0	0	

Verification :									
	1	0	1	0	1	0	0		Result
	0	1	0	1	0	1	1	→	complement and result $(43)_{10}$

Note :

Here, the magnitude of greater number is 5-bit; however, the magnitude of the result is 6-bit. Therefore, the numbers are sign-extended to 7-bits.

- For proper result we suggest to use 1 sign-bit extension to the number having greater magnitude and represent the number having smaller magnitude with extended number of bits.

Binary arithmetic - Negative numbers in 2's complement form

Case 1 (Both Positive) : Add $(28)_{10}$ and $(15)_{10}$

We have $(28)_{10} = (011100)_2$ and

$(15)_{10} = (01111)_2$

Addition of 28 and 15

			1	**1**	**1**				**Carry**
Sign-extension		**0**	0	1	1	1	0	0	$(28)_{10}$
Sign-extension	+	**0**	0	0	1	1	1	1	+ $(15)_{10}$
Sign		**0**	**1**	**0**	**1**	**0**	**1**	**1**	$(43)_{10}$

Case 2 (Smaller Negative) : Add $(28)_{10}$ and $(-15)_{10}$

We have $(011100)_2 = (28)_{10}$ and

$(01111)_2 = (15)_{10}$

$\therefore$ $(10001)_2 =$ 2's complement of 15

Addition of 28 and – 15

	1	**1**	**1**						**Carry**
Sign-extension		**0**	0	1	1	1	0	0	$(28)_{10}$
Sign-extension	+	**1**	1	1	0	0	0	1	$(-15)_{10}$
Ignore carry	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	$(13)_{10}$

Case 3 (Greater Negative) : Add $(-28)_{10}$ and $(15)_{10}$

We have $(28)_{10} = (011100)_2$ and

$(15)_{10} = (01111)_2$

$\therefore$ 2's complement of 28 $= (100100)_2$

Addition of (– 28) and (15)

				1	**1**				**Carry**
Sign-extension		**1**	1	0	0	1	0	0	$(-28)_{10}$
Sign-extension	+	**0**	**0**	0	1	1	1	1	$(15)_{10}$
		1	**1**	**1**	**0**	**0**	**1**	**1**	(–13)

Result is in 2's compliment form

Verification :		1	1	1	0	0	1	1	Result complement of result
		0	0	0	1	1	0	0	
	+							1	Add 1
		0	0	0	1	1	0	1	→ $(13)_{10}$

Case 4 (Both Negative) : Add $(-28)_{10}$ and $(-15)_{10}$

We have $(28)_{10} = (011100)_2$ and

$(15)_{10} = (01111)_2$

∴ 2's complement of 28 = $(100100)_2$

2's complement of 15 = $(10001)_2$

Addition of – 28 and – 15

	1	**1**							**Carry**
Sign-extension		**1**	1	0	0	1	0	0	**(–28)**
Sign-extension	+	**1**	**1**	1	0	0	0	1	**(–15)**
Ignore carry →	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**(–43)**

Result is in 2's complement form

Verification :		1	0	1	0	1	0	1	Result complement of result
		0	1	0	1	0	1	0	
	+							1	Add 1
		0	1	0	1	0	1	1	→ (43)

Example 2.2.3 *Perform arithmetic operation with binary numbers with negative numbers in signed 2's complementary form (– 35) + (– 40).* **Dec.-03, Marks 2**

Solution :

	0	1	0	0	0	1	1	35 in binary
	1	0	1	1	1	0	0	– 35 in 1's complement form
	1	0	1	1	1	0	1	– 35 in 2's complement form
	0	1	0	1	0	0	0	40 in binary
	1	0	1	0	1	1	1	– 40 in 1's complement form
	1	0	1	1	0	0	0	– 40 in 2's complement form
∴ (–35)		1	0	1	1	1	0	1	2's complement of – 35
+ (–40)	+	1	0	1	1	0	0	0	2's complement of – 40
	1	**0**	**1**	**1**	**0**	**1**	**0**	**1**	2's complement of 75, i.e. (–75)
Verification :	0	1	0	0	1	0	1	1	
	+							1	
	0	**1**	**0**	**0**	**1**	**0**	**1**	**0**	(+ 75)

Example 2.2.4 *Perform 1010100 – 1000100 using 1's and 2's complement.*

May-08, CSE, Marks 2

Solution : Subtraction using 1's complement

	1	0	0	0	1	0	0
	0	1	1	1	0	1	1

Subtrahend

1's complement of subtrahend

	1	**1**	**1**					**Carry**
		1	0	1	0	1	0	0	**Minuend**
+		0	1	1	1	0	1	1	**1's complement of subtrahend**
	1	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**Result**
				1	**1**	**1**	**1**		**Carry**
	1	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**Result**
+								**1**	**Add end-around carry**
		0	**0**	**1**	**0**	**0**	**0**	**0**	**Final result**

Subtraction using 2's complement

	1	0	0	0	1	0	0	**Subtrahend**
	0	1	1	1	0	1	1	**1's complement of subtrahend**
+							**1**	
	0	**1**	**1**	**1**	**1**	**0**	**0**	**2's complement of subtrahend**

	1	**1**	**1**	**1**	**1**				**Carry**
		1	0	1	0	1	0	0	**Minuend**
+		0	1	1	1	1	0	0	**2's complement of subtrahend**
Discard carry	~~1~~	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**Result**

Example 2.2.5 *Using 2's complement method perform* $(1010_2 - 1111_2)$.

Dec.-08, CSE/IT, Marks 2

Solution :

	0	0	0	0	**1's complement of 1111**
+				1	**Add 1**
	0	**0**	**0**	**1**	**2's complement of 1111**

	1	0	1	0	
+	0	0	0	1	**2's complement of 1111**
	1	**0**	**1**	**1**	**No carry, answer is negative and in 2's complement form**

Review Question

1. *Solve the following using 2's complement arithmetic.*
   a) Add 46 and 18
   b) Subtract 34 from 52

2.3 Adder and Subtractor Circuits

May-07, 08, 09, 10, 11, Dec.-06 07, 08, 09

Digital computers perform various arithmetic operations. The most basic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely,

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10_2$

The first three operations produce a sum whose length is one digit, but when the last operation is performed sum is two digits. The higher significant bit of this result is called a **carry,** and lower significant bit is called **sum**. The logic circuit which performs this operation is called a **half-adder**. The circuit which performs addition of three bits (two significant bits and a previous carry) is a **full-adder**.

2.3.1 Half Adders

The half-adder operation needs two binary inputs : augend and addend bits; and two binary outputs : sum and carry. The truth table shown in Table 2.3.1 gives the relation between input and output variables for half-adder operation.

Inputs		Outputs	
A	**B**	**Carry**	**Sum**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 2.3.1 Truth table for half-adder

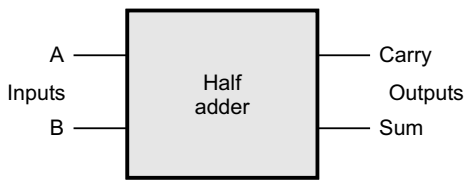


Fig. 2.3.1 Block schematic of half-adder

K-map simplification for carry and sum

For carry

A \ B	0	1
0	0	0
1	0	1

Carry = AB

For sum

A \ B	0	1
0	0	1
1	1	0

Sum = $A\overline{B} + \overline{A}B$
= $A \oplus B$

Fig. 2.3.2 Maps for half-adder

Logic diagram

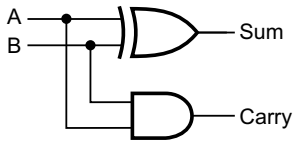


Fig. 2.3.3 Logic diagram for half-adder

Limitations of Half-Adder :

In multidigit addition we have to add two bits along with the carry of previous digit addition. Effectively such addition requires addition of three bits. This is not possible with half-adder. Hence half-adders are not used in practice.

Example 2.3.1 *Draw half adder using NAND gates.*

Solution : For half adder :

$$\text{Sum} = A\overline{B} + \overline{A}B \qquad C_{out} = AB$$

$$= \overline{\overline{A\overline{B} + \overline{A}B}} \qquad = \overline{\overline{AB}}$$

$$= \overline{\overline{AB}} = \overline{A\overline{B}} \cdot \overline{\overline{A}B}$$

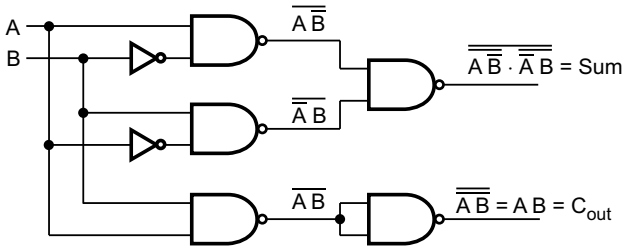


Fig. 2.3.4

2.3.2 Full Adders

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by A and B, represent the two significant bits to be added. The third input C_{in}, represents the carry from the previous lower significant position. The truth table for full-adder is shown in Table 2.3.2.

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 2.3.2 Truth table for full-adder

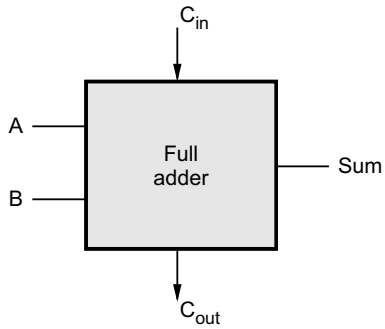
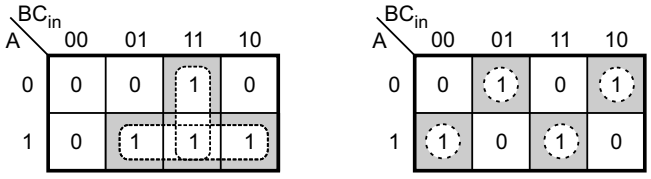


Fig. 2.3.5 Block schematic of full-adder

K-map simplification for carry and sum

For carry (C_{out}) **For sum**



$C_{out} = AB + A\,C_{in} + B\,C_{in}$ $\quad$ $Sum = \overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C}_{in} + A\overline{B}\,\overline{C}_{in} + ABC_{in}$

Fig. 2.3.6 Maps for full-adder

Logic diagram

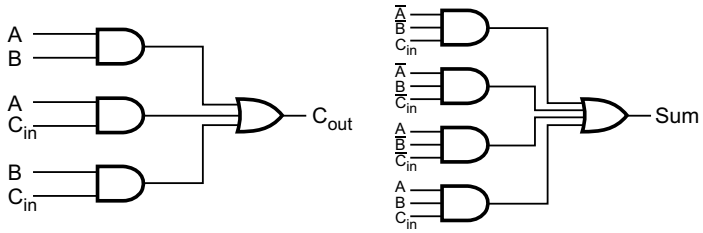


Fig. 2.3.7 Sum of product implementation of full-adder

The Boolean function for sum can be further simplified as follows :

$$\begin{aligned} Sum &= \overline{A}\ \overline{B}\,C_{in} + \overline{A}\,B\,\overline{C}_{in} + A\,\overline{B}\ \overline{C}_{in} + A\,B\,C_{in} \\ &= C_{in}\,(\overline{A}\,\overline{B} + AB) + \overline{C}_{in}\ (\overline{A}\,B + A\,\overline{B}) \ = \ C_{in}\,(A \odot B) + \overline{C}_{in}\ (A \oplus B) \\ &= C_{in}\,(\overline{A \oplus B}) + \overline{C}_{in}\ (A \oplus B) = C_{in} \oplus (A \oplus B) \end{aligned}$$

With this simplified Boolean function circuit for full-adder can be implemented as shown in the Fig. 2.3.8.

A full-adder can also be implemented with two half-adders and one OR gate, as shown in the Fig. 2.3.9. The sum output from the second half-adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

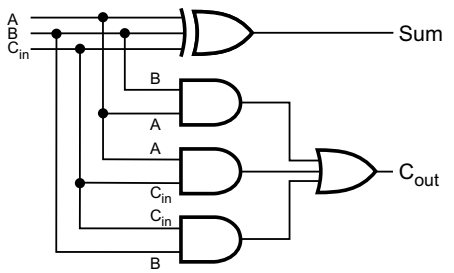


Fig. 2.3.8 Implementation of full-adder

$$\begin{aligned} Cout &= AB + A\,C_{in} + B\,C_{in} = AB + A\,C_{in}\,(B + \overline{B}) + B\,C_{in}\,(A + \overline{A}) \\ &= AB + ABC_{in} + A\,\overline{B}\,C_{in} + ABC_{in} + \overline{A}\,BC_{in} \\ &= AB\,(1 + C_{in} + C_{in}) + A\,\overline{B}C_{in} + \overline{A}\,BC_{in} \\ &= AB + A\overline{B}C_{in} + \overline{A}BC_{in} = AB + C_{in}\,(A\overline{B} + \overline{A}B) = AB + C_{in}\,(A \oplus B) \end{aligned}$$

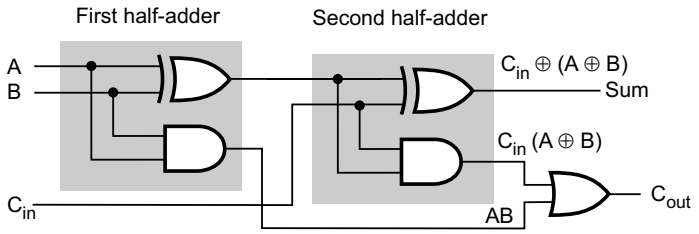


Fig. 2.3.9 Implementation of a full-adder with two half-adders and an OR gate

2.3.3 Parallel Adder

A single full-adder is capable of adding two one-bit numbers and an input carry. In order to add binary numbers with more than one bit, additional full-adders must be employed. A n-bit, parallel adder can be constructed using number of full adder circuits connected in parallel.

Fig. 2.3.10 shows the block diagram of n-bit parallel adder using n number of full-adder circuits connected in cascade, i.e. the carry output of each adder is connected to the carry input of the next higher-order adder

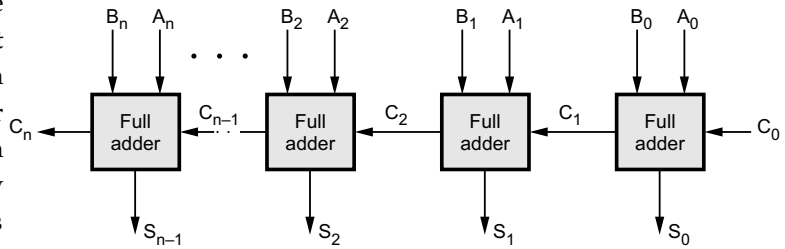


Fig. 2.3.10 Block diagram of n-bit parallel adder

It should be noted that either a half-adder can be used for the least significant position or the carry input of a full-adder is made 0 because there is no carry into the least significant bit position.

2.3.4 Parallel Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements. The subtraction A – B can be done by taking the 2′s complement of B and adding it to A.

The 2′s complement can be obtained by taking the 1′s complement and adding one to the least significant pair of bits. The 1′s complement can be implemented with inverters and a one can be added to the sum through the input carry to get 2's complement, as shown in the Fig. 2.3.11.

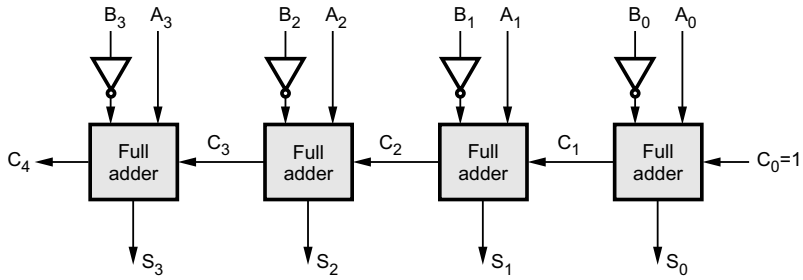


Fig. 2.3.11 4-bit parallel subtractor

Example 2.3.2 *Design a combinational circuit whose input is a 4-bit number and whose output is the 2′s complement of the input number.*

Solution :

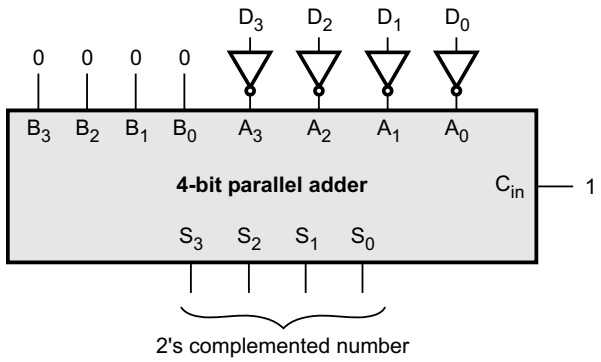


Fig. 2.3.12

2.3.5 Addition / Subtraction Logic Unit

Fig. 2.3.13 shows hardware to implement integer addition and subtraction. It consists of n-bit adder, 2's complement circuit, overflow detector logic circuit and AVF (overflow flag). Number a and number b are the two inputs for n-bit adder. For subtraction, the subtrahend (number from B register) is converted into its 2's complement form by making Add/Subtract control signal to the logic one. When Add/Subtract control signal is one, all bits of number b are complemented and carry zero (C_0) is set to one. Therefore n-bit adder gives result as $R = a + \overline{b} + 1$, where $\overline{b}+1$ represents 2's complement of number b.

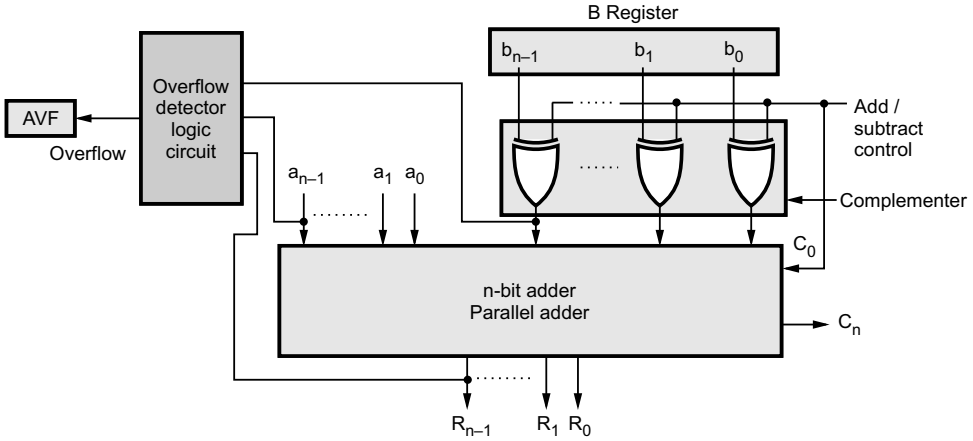


Fig. 2.3.13 Hardware for integer addition and subtraction

2.3.6 Overflow in Integer Arithmetic

When adding signed numbers, a carry bit beyond the end of the word does not serve as the overflow indicator. If we add the numbers + 7 and + 3 in a 4-bit adder, the output is 1010, which is the code of – 6, a wrong result. In this case, carry bit from the MSB position is 0. Similarly, if we add – 5 and – 6, we get output = + 5, another error. In this case carry bit from the MSB position is 1. One thing we can surely say that, the addition of numbers with different signs cannot cause overflow, because the absolute value of the sum is always smaller than the absolute

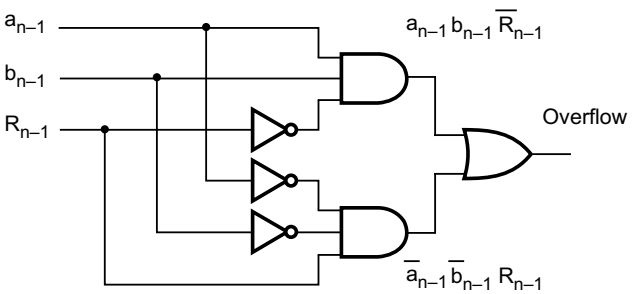


Fig. 2.3.14 Overflow detector logic circuit

value of one of the two operands. From above discussion we can conclude following points :

1. Overflow can occur only when adding two numbers that have the same sign.
2. The carry bit from the MSB position is not a sufficient indicator of overflow when adding signed numbers.
3. When both operands a and b have the same sign, an overflow occurs when the sign of result does not agree with the signs of a and b. The logical expression to detect overflow can be given as

$$\text{Overflow} = a_{n-1}\, b_{n-1}\, \overline{R_{n-1}} + \overline{a_{n-1}}\, \overline{b_{n-1}}\, R_{n-1}$$

where a_{n-1} = MSB of number a

b_{n-1} = MSB of number b

R_{n-1} = MSB of the result

Example 2.3.3 *Give means to identify on whether or not an overflow has occurred in 2's complement addition or subtraction operations. Take one example for each possible situation and explain. Assume 4-bit registers.*

Solution : Case 1 : Both numbers positive

	1	**1**	**1**		**Carry**
	0	1	1	1	**(+ 7)**
+	0	0	1	1	**(+ 3)**
	1	**0**	**1**	**0**	**Result : 2's complement of 6**

Fig. 2.3.15

Result is – 6 ; it is wrong due to overflow.

Case 2 : Both numbers negative

1					**Carry**
	1	0	1	1	**2's complement of 5, i.e. (– 5)**
+	1	1	0	0	**2's complement of 4, i.e. (– 4)**
	0	**1**	**1**	**1**	**(+ 7)**

Fig. 2.3.16

Result is + 7 ; it is wrong due to overflow.

2.3.7 Addition and Subtraction of Signed-magnitude Data

The Fig. 2.3.17 shows the hardware and flowchart showing the procedure for addition and subtraction with signed-magnitude data. Initially, signs of two numbers are compared by an exclusive-OR gate. When output of EX-OR gate is 0, the signs are same if it is 1, the signs are different. For an add operation with equal signs or for a subtraction operation with different signs magnitudes of numbers are added using micro-operation EA ← A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. It is then transfer into the add-overflow flip-flop AVF.

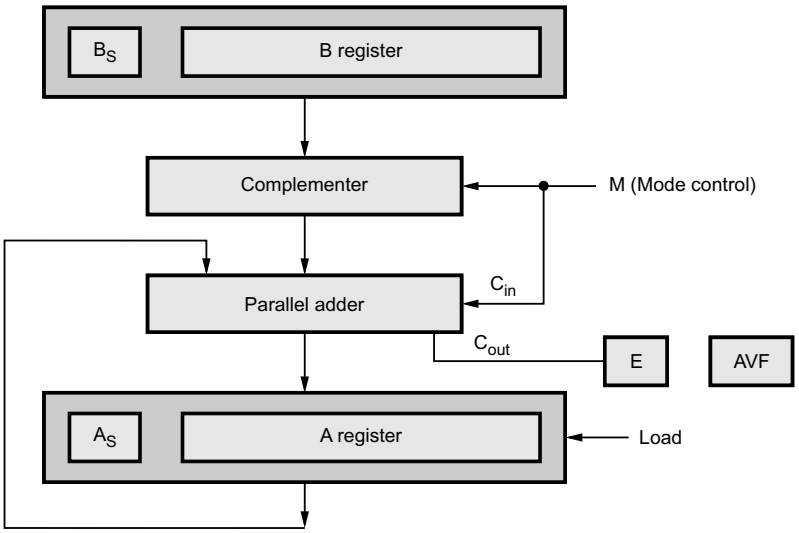


Fig. 2.3.17 (a) Hardware for signed-magnitude addition and subtraction

The two magnitudes are subtracted if the signs are different for an add operation or same for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. Since magnitude of numbers are subtracted there is no overflow and AVF is clear to 0. A 1 in E indicates that $A \geq B$ and the number in A is the true result. If result is zero, the sign A_s is made positive to avoid a negative zero. A 0 in E indicates that A < B and the result is in A is not true. Here, it is necessary to take 2's complement of the result. The 2's complement of result can be performed using micro-operation $A \leftarrow \overline{A} + 1$. The micro-operation $A \leftarrow \overline{A} + 1$ is carried out using operations :

$A \leftarrow \overline{A}$, $A \leftarrow A + 1$, $A_s \leftarrow \overline{A}_s$

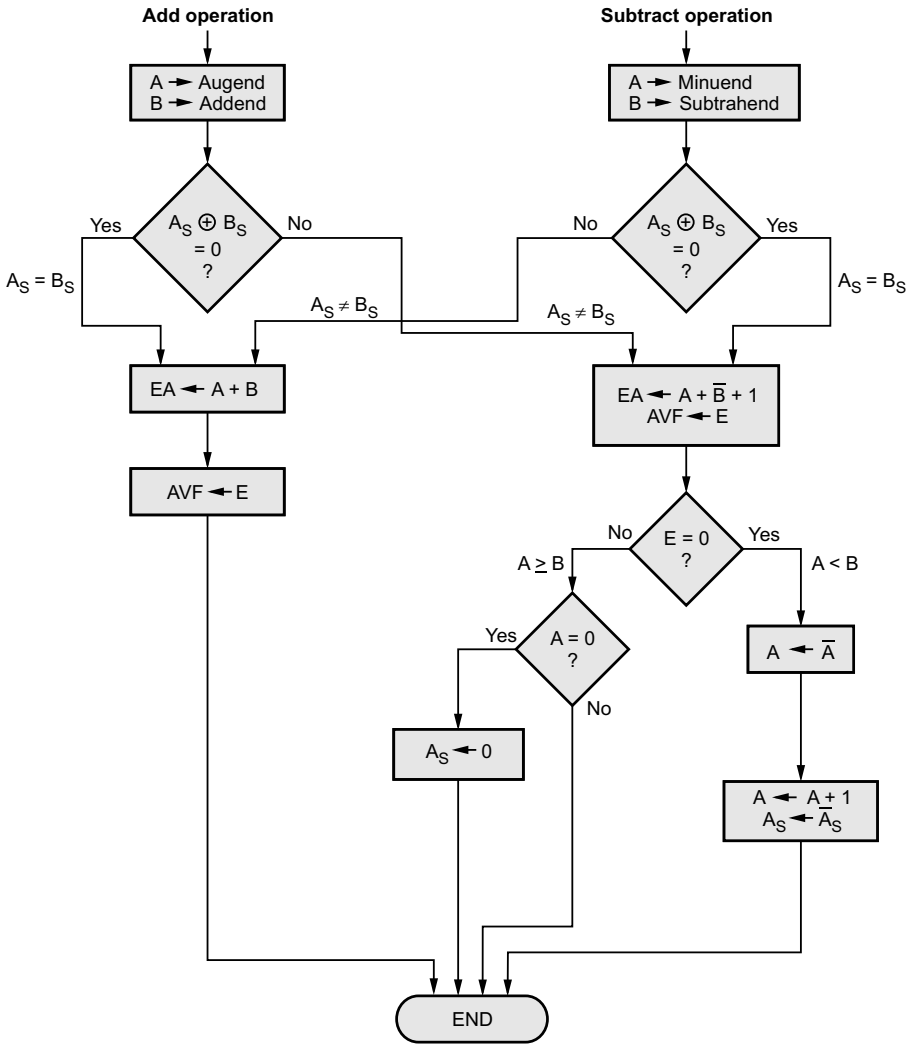


Fig. 2.3.17 (b) Flowchart for add and subtract operations

Review Questions

1. *What is half adder? Design a half adder as a two-level AND-OR circuit and show how to implement a full adder using two half adders and a external logic gate.*
   May-09, Marks 8
2. *Draw the half adder circuit.*
   May-09, Marks 2
3. *Write the logic equations of a binary half a adder.*
   May-11, Marks 2
4. *Draw the symbolic representation of the full-adder and give the expression for the sum.*
   Dec.-06, Marks 2

5. *Draw a full-adder circuit and give the truth table.* **May-07, Marks 2**
6. *Draw the full adder circuit using two half adders.* **Dec.-07, Marks 2**
7. *Design the full adder circuit.* **May-09, Marks 16**
8. *Draw and explain a block diagram of parallel adder.*
9. *What is a ripple carry adder ?* **May-07, Marks 2**
10. *Draw and explain the block diagram of parallel subtractor.*
11. *Explain the working of parallel subtractor.*
12. *Explain the hardware for signed-magnitude addition subtraction with block diagram.* **Dec.-07, Marks 10**
13. *Design a 4-bit binary adder/subtractor and explain its functions.* **May-08, Marks 8**
14. *With examples explain the data transfer, logic and program control instructions?* **May-10, Marks 16**
15. *Indicate how an overflow is detected ?* **Dec.-08, Marks 10**
16. *Define overflow rule in addition.* **Dec.-09, Marks 2**
17. *Give the block diagram of the hardware implementation of addition and subtraction of signed number and explain the operations with flowchart.* **May-07, Marks 10**
18. *Explain the procedure for addition and subtraction with signed-magnitude data with the help of flowchart.*

2.4 Look Ahead Carry Adders

Dec.-06, 07, 08, 10, May-06, 07, 08, 09, 10, 11, 12

The n-bit adder ripple carry adder discussed in the last section is implemented using full-adder stages. In which the carry output of each full-adder stage is connected to the carry input of the next higher-order stage. Therefore, the sum and carry outputs of any stage cannot be produced until the input carry occurs; this leads to a time delay in the addition process. This delay is known as **carry propagation delay**, which can be best explained by considering the following addition.

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0 \end{array}$$

Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position (stage), produces a carry into the third position. The latter carry, when added to the bits of the third position, produces a carry into the last position. The key thing to notice in this example is that the sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous positions. This means that, adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in each full-adder. For

example, if each full-adder is considered to have a propagation delay of 30 ns, then S_3 will not reach its correct value until 90 ns after LSB carry is generated. Therefore, total time required to perform addition is $90 + 30 = 120$ ns.

Obviously, this situation becomes much worse if we extend the adder circuit to add a greater number of bits. If the adder were handling 16-bit numbers, the carry propagation delay could be 480 ns.

Generally, carry propagation delay and sum propagation delay are measured in terms of gate delays. Looking at Fig. 2.3.8 we can notice that for full-adder C_{out} requires two gate delays and sum requires only one gate delay. When we connect such full adder circuits in cascade to generate n-bit ripple adder as shown in the Fig. 2.3.10, C_{n-1} is available in 2(n–1) gate delays, and S_{n-1} is correct one XOR gate delay later. The final carry-out, C_n is available after 2n gate delays. Thus for 4-bit ripple adder C_4 is available after 8(2×4) gate delays, C_3 is available in 6[2(4–1)] gate delays and S_3 is available in 7 gate delays.

One method of speeding up this process by eliminating inter stage carry delay is called **lookahead-carry addition.** This method utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions : carry generate and carry propagate.

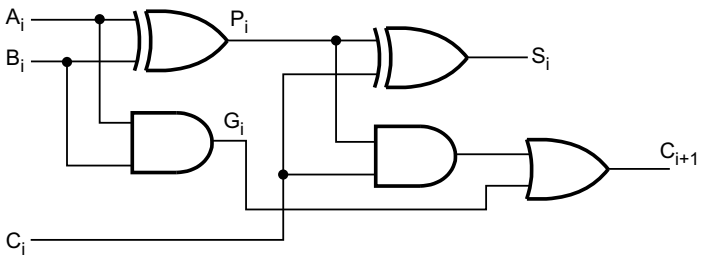


Fig. 2.4.1 Full adder circuit

Consider the circuit of the full adder shown in Fig. 2.4.1. Here, we define two functions : **carry generate** and **carry propagate.**

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i \quad \text{(Refer Appendix-A for details.)}$$

The output sum and carry can be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i is called a carry generate and it produces on carry when both A_i and B_i are one, regardless of the input carry. P_i is called a carry propagate because it is term associated with the propagation of the carry from C_i to C_{i+1}. Now C_{i+1} can be expressed as a sum of products function of the P and G outputs of all the preceding stages. For example, the carriers in a four stage carry-lookahead adder are defined as follows :

$$C_1 = G_0 + P_0 C_{in}$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$

$$C_4 = G_3 + P_3 C_3$$

$$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

Fig. 2.4.2 shows the general form of a carry-lookahead adder circuit designed in this way.

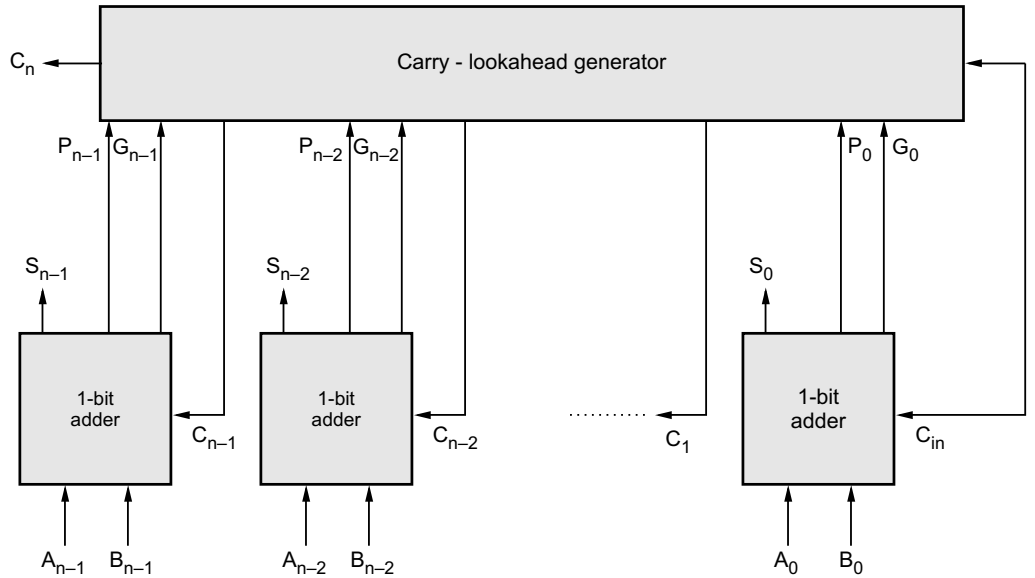


Fig. 2.4.2 General form of a carry-lookahead adder circuit

We can further simplify the design by noting that the sum equation of stage i

$S_i = A_i \oplus B_i \oplus C_i$ as $S_i = P_i \oplus G_i \oplus C_i$

Combining equations for P and G, the carry-lookahead equations, and the modified sum equation, we obtain the 4-bit carry-lookahead adder as shown in Fig. 2.4.3. Looking at Fig. 2.4.3 we can easily note that one gate delay is needed to develop all P_i and G_i signals, followed by two gate delays in the AND-OR circuit for C_{i+1}. After a further XOR gate delay, all sum bits are available. Therefore, independent of n, the n-bit addition process requires only 4 gate delays and all carries can be obtained within 3 gate delays after the signals A, B and C_{in} are applied. In

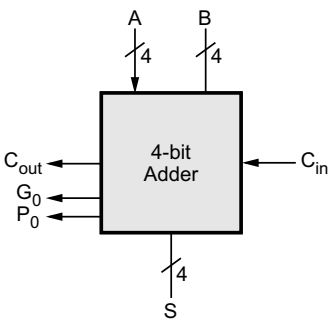


Fig. 2.4.3

comparison note that a 4-bit ripple carry adder requires 7 gate delays for S_3 and 8 gate delays for C_4.

A complete 4-bit carry-lookahead adder in a block form is shown in the Fig. 2.4.3.

We can cascade such 4-bit carry-lookahead adders to form a 16-bit or 32-bit adder. We require four 4-bit carry-lookahead adders to form 16-bit carry-lookahead adder and eight 4-bit carry-lookahead adders to form 32-bit carry-lookahead adder. In 32-bit adder, the carry out C_4 form the low-order 4-bit adder is available 3 gate delays after the input operands A, B and C_0 are applied to the 32-bit adder. Then C_8 is available at the output of the second adder after a further 2 gate delays, C_{12} is available after a further 2 gate delays, and so on. Finally, C_{28} the carry-in to the high-order 4-bit adder, is available after a total of (6 × 2) + 3 = 15 gate delays, C_{32} is available after 2 more gate delays, i.e. after 17 gate delays and S_{31} is available after 3 gate delays, i.e. 18 gate delays. These gate delays are very less compared to total delays of 63 and 64 for S_{31} and C_{32} if ripple-carry adder is used.

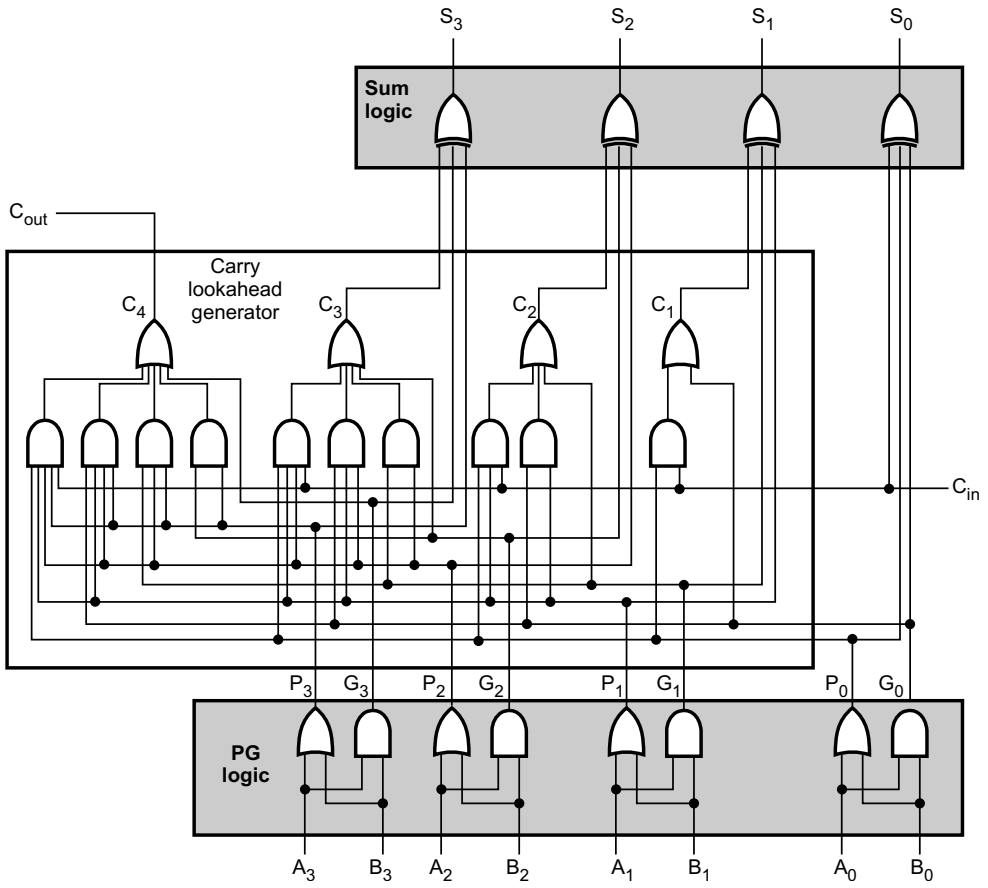


Fig. 2.4.4

Multilevel Generate and Propagate Functions

In the 32-bit adder just discussed, the carriers C_4, C_8, C_{12}, ... ripple through the 4-bit adder blocks with two gate delays per clock. This is analogous to the way that individual carries ripple through each bit stage in a ripple-carry adder. By using multi-level block generate and propagate functions, it is possible to use the lookahead approach to develop the carries C_4, C_8, C_{12}, ... in parallel, in a multi-level carry-lookahead circuit. The Fig. 2.4.5 shows a 16-bit adder implemented using four 4-bit adder blocks. Here, blocks provide new output functions defined as G^I_K and P^I_K, where K = 0 for the first 4-bit block, K = 1 for the second 4-bit block and so on. In the first block,

$$P^I_0 = P_3P_2P_1P_0$$

and

$$G^I_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

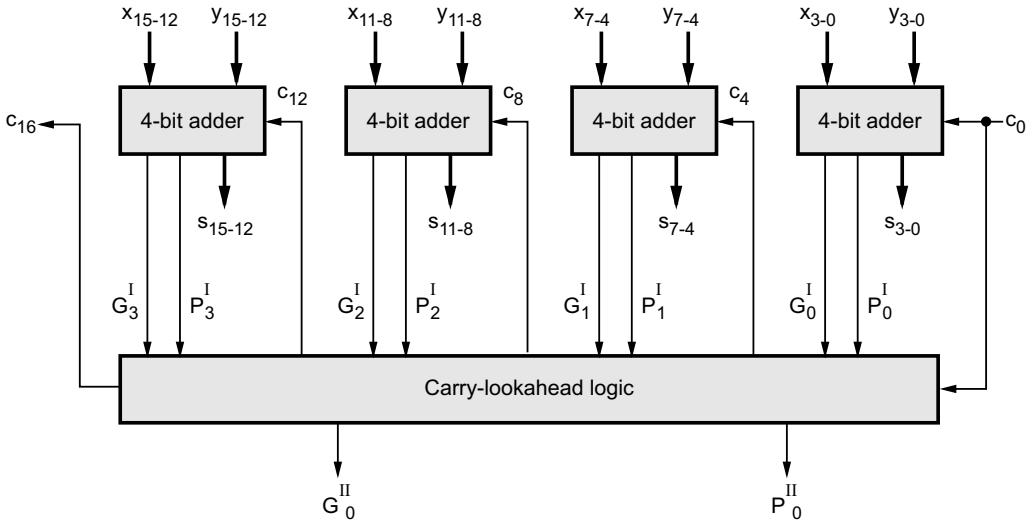


Fig. 2.4.5 16-bit carry-lookahead adder built from 4-bit adders

Therefore, we can use first-level G_i and P_i functions to determine whether bit stage i generates or propagates a carry, and we can use the second level G^I_K and P^I_K functions to determine whether block K generates or propagates a carry. With these new functions available, it is not necessary to wait for carries to ripple through the 4-bit blocks. Looking at Fig. 2.4.5, we can determine C_{16} as

$$C_{16} = G^I_3 + P^I_3G^I_2 + P^I_3P^I_2G^I_1 + P^I_3P^I_2P^I_1G^I_0 + P^I_3P^I_2P^I_1P^I_0C_0$$

The above expression is identical in form to the expression for C_4; only variable names are different. Therefore, the structure of the carry-lookahead circuit in the Fig. 2.4.5 is identical to the carry-lookahead circuit shown in the Fig. 2.4.4. However, it is important to note that the carries C_4, C_8, C_{12} and C_{16} generated internally by the 4-bit adder blocks are not needed in the Fig. 2.4.5 because they are generated by the multi-level carry-lookahead circuits.

Let see the delay in producing outputs from the 16-bit carry-ahead adder. The delay in developing the carries produced by the carry-lookahead circuits is two gate delays more than the delay needed to develop the G_K^I and P_K^I functions. The G_K^I and P_K^I functions require two gate delays and one gate delay, respectively, after the generation of G_i and P_i. Therefore, all carries produced by the carry-lookahead circuits are available 5 gate delays after A, B and C_0 are applied as inputs. The carry C_{15} is generated inside the higher-order 4-bit block in the Fig. 2.4.5 in two gate delays after C_{12} followed by S_{15} in one further gate delay. Therefore, S_{15} is available after 8 gate delays. These delays, 5 gate delays for C_{16} and 8 gate delays for S_{15} are less as compared to 9 and 10 gate delays for C_{16} and S_{15} in cascaded 4-bit carry-lookahead adder blocks, respectively.

We can cascade two 16-bit adders to implement 32-bit adder. Here, only two more gate delays are required to get C_{32} and S_{31} and C_{16} and S_{15}, respectively. Therefore, C_{32} is available after 7 gate delays and S_{31} is available after 10 gate delays. These delays are less compared to 18 and 17 gate delays for the same outputs if the 32-bit adder is built from a cascade of eight 4-bit adders.

If we go for third level then we can built 64-bit using four 16-bit adders. Delay through this adder will be 12 gate delays for S_{63} and 7 gate delays for C_{64}.

Review Questions

1. *Explain in detail the principle of carry-look-ahead adder.* **Dec.-06, 07, Marks 12, May-07, 08, 09, 11 Marks 8**
2. *Show how 16-bit CLAs can be constructed from 4-bit adders.* **Dec.-06, 07, Marks 12, May-07, 08, 09, 11 Marks 8**
3. *What is the disadvantage in using a ripple carry adder ?* **Dec.-06, Marks 4**
4. *With a neat sketch, explain in detail about logic design for fast adders.* **Dec.-07, Marks 16**
5. *How do you ascertain an adder as "fast" ?* **Dec.-08, Marks 2**
6. *Design a 4-bit fast adder and explain its function in detail.* **May-10, Marks 8**
7. *Explain in detail the principle of carry-look-ahead adder. Show how 16-bit CLAs can be constructed from 4-bit adders.* **May-06, Marks 12**

8. *Draw the diagram of a carry look ahead adder and explain the carry look ahead principle.* **Dec.-06, Marks 12, Dec.-10, Marks 16**
9. *What is a ripple carry adder ?* **May-07, Marks 2**
10. *Discuss the principle of operation of carry-look ahead adders.* **May-07, Marks 8**
11. *Design a 4-bit carry-lookahead adder and explain its operation with an example.* **Dec.-07, Marks 8**
12. *With a neat sketch, explain in detail about logic design for fast adders.* **Dec.-07, Marks 16**
13. *Design a 4-bit carry-lookahead adder.* **May-08, Marks 16**
14. *Describe the design of 4-bit carry-lookahead adder.* **May-09, Marks 8**
15. *Design a 4-bit full adder with look-ahead carry.* **May-11, Marks 8**
16. *Draw and explain the block diagram used to perform carry look ahead addition with the necessary equations.* **May-12, Marks 16**

2.5 Multiplication

Dec.-08, May-07

The multiplication is a complex operation than addition and subtraction. It can be performed in hardware or software. A wide variety of algorithms have been used in various computers. For simplicity we will first see the multiplication algorithms for unsigned (positive) integers, and then we will see the multiplication algorithm for signed numbers.

Fig. 2.5.1 shows the usual algorithm for multiplying positive numbers by hand. Looking at this algorithms we can note following points :

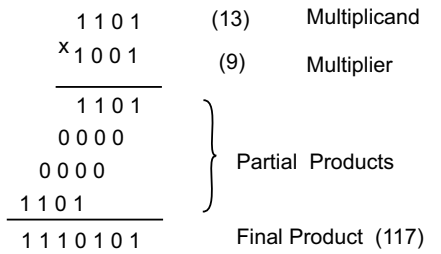
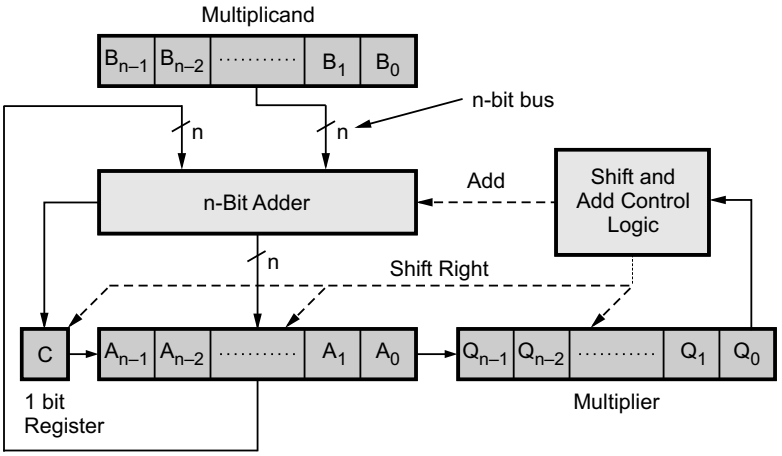


Fig. 2.5.1 Manual multiplication algorithm

- Multiplication process involves generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
- In the binary system the partial products are easily defined. When the multiplier bit is 0, the partial product is 0, and when the multiplier is 1, the partial product is the multiplicand.
- The final product is produced by summing the partial products. Before summing operation each successive partial product is shifted one position to the left relative to the preceding partial product, as shown in the Fig. 2.5.1.
- The product of two n-digit numbers can be accommodated in 2n digits, so the product of the two 4-bit numbers in fits into 8-bits.

Fig. 2.5.2 shows the implementation of manual multiplication approach. It consists of n-bit binary adder, shift and add control logic and four registers, A, B, C, and Q. As shown in the Fig. 2.5.2 multiplier and multiplicand are loaded into register Q and register B, respectively, and C are initially set to 0.



Note : Dotted lines indicate control signals

Fig. 2.5.2 Hardware implementation of unsigned binary multiplication

Multiplication Operation Steps

1. Bit 0 of multiplier operand (Q_0 of Q register) is checked.
2. If bit 0 (Q_0) is one then multiplicand and partial product are added and all bits of C, A and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If bit 0 (Q_0) is 0, then no addition is performed, only shift operation is carried out.
3. Steps 1 and 2 are repeated n times to get the desired result in the A and Q registers.

A flowchart for multiplication operation is shown in Fig. 2.5.3.

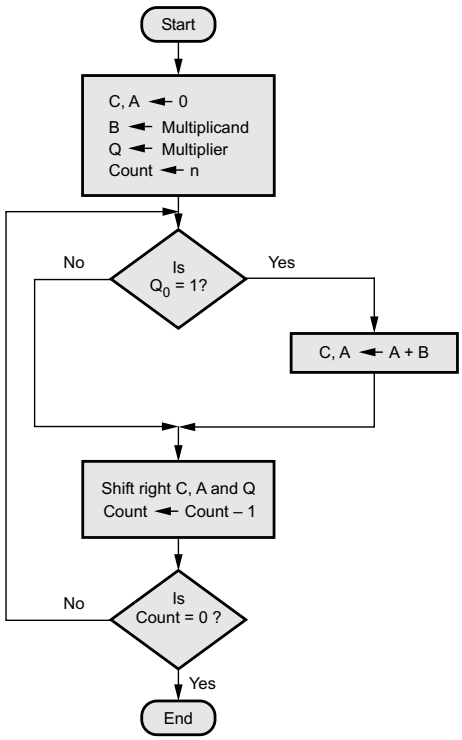


Fig. 2.5.3 Flowchart for multiplication operation

Let us see one example.

Consider 4-bit multiplier and multiplicand :

Multiplicand = 1 1 0 1 and Multiplier = 1 0 1 1

Fig. 2.5.4 shows operations involved and their results in the multiplication process.

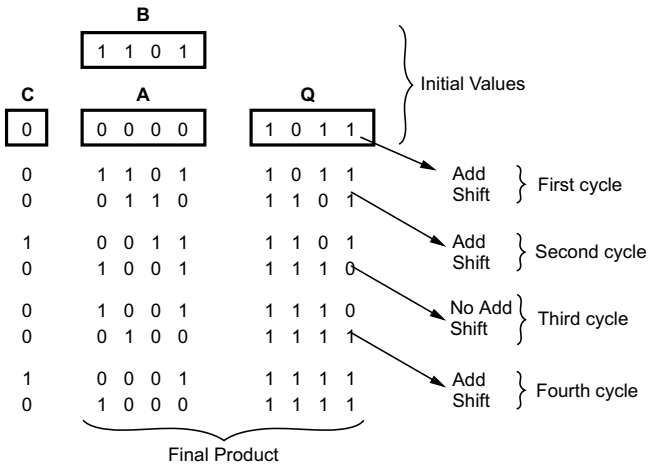


Fig. 2.5.4 Multiplication process

Review Questions

1. *Explain an algorithm to multiply two positive numbers. Also discuss the realization of a multiplier to implement the same.* **Dec.-08, Marks 16**
2. *Design a multiplier that multiplies two 4-bit numbers.* **May-07, Marks 8**

2.6 Robertson Algorithm

Dec.-03, 11

2.6.1 Robertson Algorithm for 2's Multiplier

The multiplication process which we have just discussed will not work for signed numbers. For a negative multiplier, a straightforward solution is to form the 2's complement of both the multiplier and multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. Suppose that Q is the multiplier operand in a shift and add multiplication to compute $P = Q \times B$ for twos-complement numbers. James E. Robertson suggests that we can use an unsigned multiplication technique like that illustrated in Fig. 2.5.2 with one change : when multiplying by the sign bit, perform subtraction rather than addition in the final step if a minus sign $Q_0 = 1$ is encountered.

The Fig. 2.6.1 shows the hardware for implementation of Robertson algorithm for twos complement multiplication. As shown in the Fig. 2.6.1 register B stores multiplicand and register Q stores multiplier. The registers A · Q forms a right shift register.

The n-bit adder performs addition of two inputs. One input is the A register and other input is multiplicand. In case of addition, $\overline{\text{Add}}$/sub line is 0, therefore $C_{in} = 0$ and multiplicand is directly applied as a second input to the n-bit adder. In case of

subtraction, $\overline{Add}$/sub line is 1, therefore C_{in}=1 and multiplicand is complemented and then applied to the n-bit adder. As a result, the 2's complement of multiplicand is added in the A register.

The shift, add and subtract control logic checks bit Q_0. If Q_0 = 1, it generates control signals for the addition of A and B and then for shift right operation of A · Q register. This shift operation preserves the sign bit and hence it is known as arithmetic shift. However, in the final step if Q_0 = 1, it generates control signals for subtraction (A ← A – B) instead of addition. The Table 2.6.1 shows truth table for shift, add and subtract control logic.

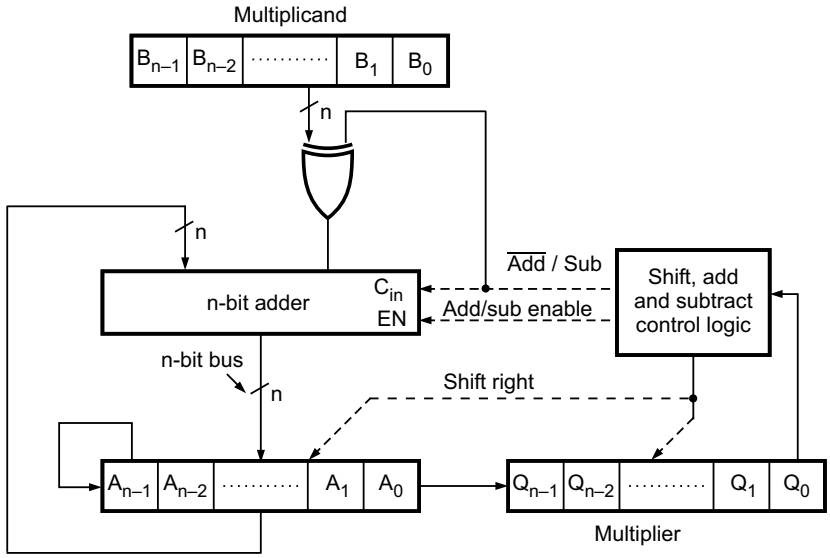


Fig. 2.6.1 Hardware implementation of Robertson algorithm for 2's complement multiplication

Step	Q_0	$\overline{Add}$/sub	Add/sub enable	Shift
X	0	X	0	1
Not final	1	0	1	1
Final	1	1	1	1

Table 2.6.1 Truth table for shift, add and subtract control logic

Algorithm :

1. A ← 0, B ← Multiplicand Q ← Multiplier, and count n.
2. If Q_0 = 1 then perform A ← A + B.
3. A_{n-1} ← A_{n-1}, shift right register A · Q by 1-bit count ← count – 1.

4. If count > 1 Repeat steps 2 and 3 ; otherwise if $Q_0 = 1$ then perform $A \leftarrow A - B$ else perform $A \leftarrow A + B$.
5. $A_{n-1} \leftarrow A_{n-1}$, shift right register $A \cdot Q$ by 1-bit.

Flowchart

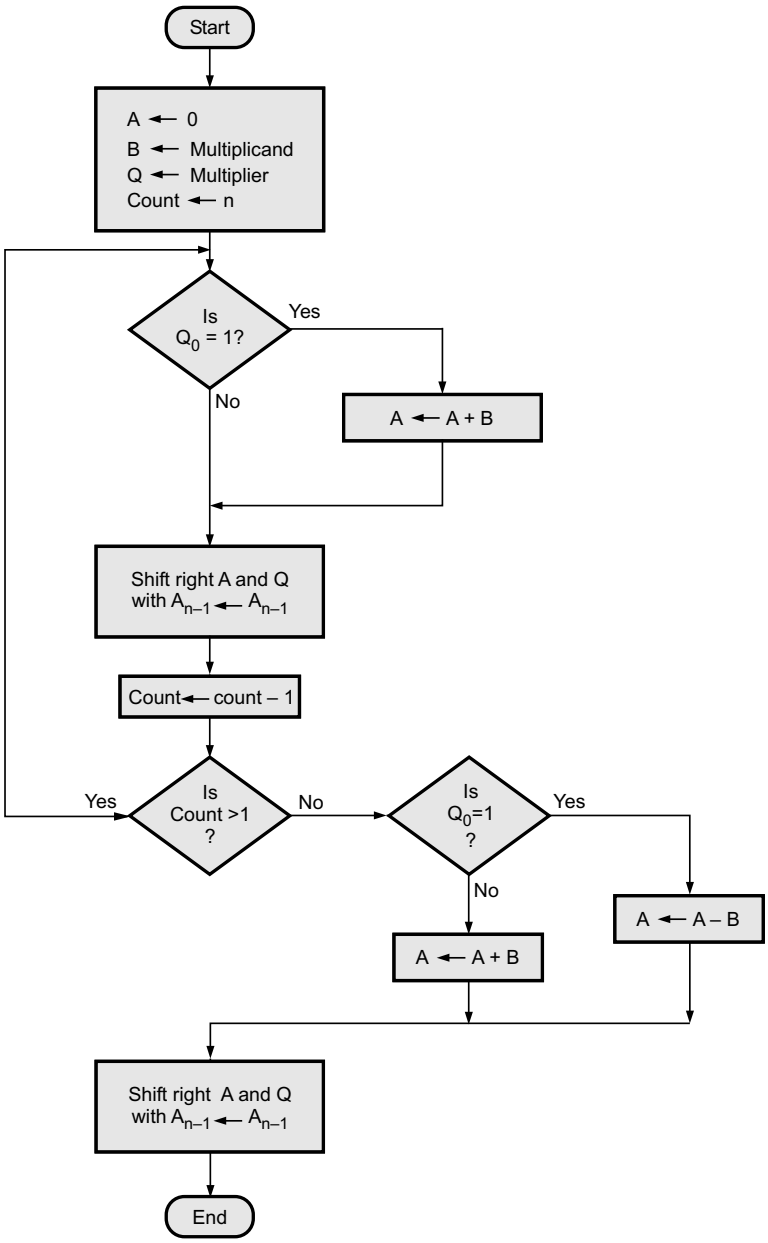


Fig. 2.6.2

Let us see the multiplication of 4-bit numbers, 5 and 4 with all possible combinations.

CASE 1 : Both positive (5 × 4)

Multiplicand (B) ← 0 1 0 1 (5) Multiplier (Q) ← 0 1 0 0 (4)			
Steps	**A**	**Q**	**Operation**
	0 0 0 0	0 1 0 **0**	Initial
Step 1	0 0 0 0	0 0 1 **0**	Shift right
Step 2	0 0 0 0	0 0 0 **1**	Shift right
Step 3	0 1 0 1	0 0 0 1	A ← A + B (A ← 0 0 0 0 + 0 1 0 1)
	0 0 1 0	1 0 0 **0**	Shift right
Step 4 (Final)	0 0 0 1	0 1 0 0	Shift right
Result : 0 0 0 1		0 1 0 0 = + 20	

CASE 2 : Negative multiplier (5 × – 4)

Multiplicand (B) ← 0 1 0 1 (5) Multiplier (Q) ← 1 1 0 0 (– 4)			
Steps	**A**	**Q**	**Operation**
	0 0 0 0	1 1 0 **0**	Initial
Step 1	0 0 0 0	0 1 1 **0**	Shift right
Step 2	0 0 0 0	0 0 1 **1**	Shift right
Step 3	0 1 0 1	0 0 1 1	A ← A + B (A ← 0 0 0 0 + 0 1 0 1)
	0 0 1 0	1 0 0 **1**	Shift right
Step 4 (Final)	1 1 0 1	1 0 0 1	A ← A – B (A ← 0 0 1 0 + 1 0 1 1)
	1 1 1 0	1 1 0 0	Shift right
Result : 1 1 1 0		1 1 0 0 = – 20	

CASE 3 : Negative multiplicand (– 5 × 4)

Multiplicand (B) ← 1 0 1 1 (– 5) Multiplier (Q) ← 0 1 0 0 (4)			
Steps	**A**	**Q**	**Operation**
	0 0 0 0	0 1 0 **0**	Initial
Step 1	0 0 0 0	0 0 1 **0**	Shift right
Step 2	0 0 0 0	0 0 0 **1**	Shift right

Step 3	1 0 1 1	0 0 0 1	A ← A + B (A ← 0 0 0 0 + 1 0 1 1)
	1 1 0 1	1 0 0 **0**	Shift right
Step 4 (Final)	1 1 1 0	1 1 0 0	Shift right
Result : 1 1 1 0		1 1 0 0 = – 20	

CASE 4 : Both negative (– 5 × – 4)

Multiplicand (B) ← 1 0 1 1 (– 5) Multiplier (Q) ← 1 1 0 0 (– 4)			
Steps	**A**	**Q**	**Operation**
	0 0 0 0	1 1 0 **0**	Initial
Step 1	0 0 0 0	0 1 1 **0**	Shift right
Step 2	0 0 0 0	0 0 1 **1**	Shift right
Step 3	1 0 1 1	0 0 1 1	A ← A + B (A ← 0 0 0 0 + 1 0 1 1)
	1 1 0 1	1 0 0 **1**	Shift right
Step 4 (Final)	0 0 1 0	1 0 0 1	A ← A – B (A ← 1 1 0 1 + 0 1 0 1)
	0 0 0 1	0 1 0 0	Shift right
Result : 0 0 0 1		0 1 0 0 = + 20	

2.6.2 Robertson Algorithm for 2's Complement Fraction

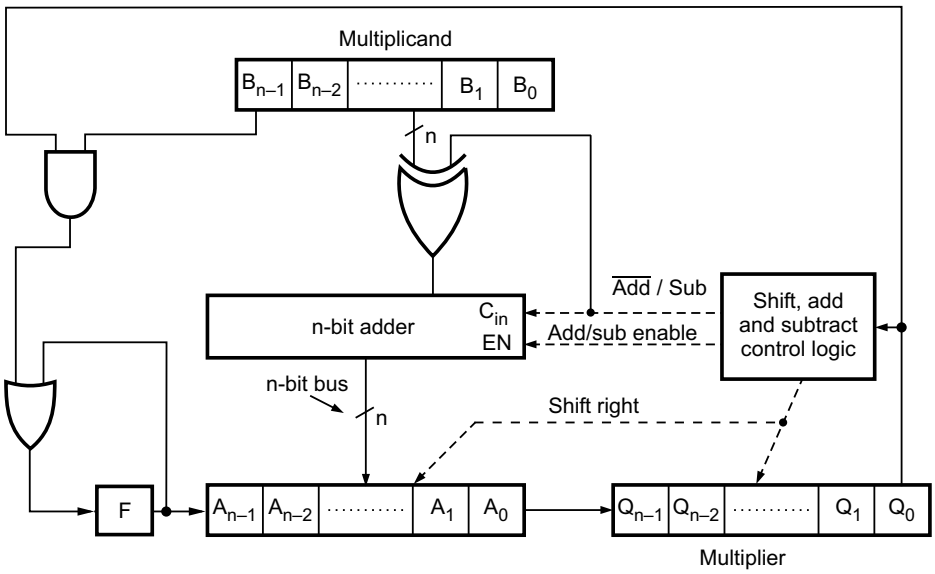
Consider the task of multiplying two n-bit binary fractions using twos-complement code. The multiplier has a similar register level structure to that in Fig. 2.6.1, with registers A, B and Q storing the various operands and A · Q forming a right shift register.

Each addition/subtraction step can be performed in the usual twos-complement fashion by treating the sign bits like any other and ignoring overflow. Care is needed in the shift step to ensure that the correct new value is placed in the accumulator's sign bit A [n – 1]. This value must be a leading 0 if the current partial product in A · Q is positive or zero, and 1 if it is negative. A flip-flop F is used to control the values assigned to A [n – 1]. F is initially set to 0, and is subsequently defined by

F = (Initial value of B[n – 1] and current value of Q[0]) or F

Here, initial value of B[n – 1] represents sign of multiplicand. Thus F is set to 1 if multiplicand in negative and at least one non zero Q[0] is encountered. Once set to 1, it remains at that value.

The Fig. 2.6.3 shows the hardware implementation of Robertson algorithm for 2's complement fractions.



● NOTE : Dotted lines indicate control signals

Fig. 2.6.3 Hardware implementation of Robertson algorithm for 2's complement fraction

Algorithm :

1. A ← 0, B ← Multiplicand Q ← Multiplier, and count ← n.
2. If Q_0 = 1 then perform A ← A + B.
3. Shift right register F, A · Q by 1-bit.

   F ← B [n – 1] AND Q[0] OR F and

   count ← count – 1.
4. If count > 1 Repeat steps 2 and 3 ; otherwise if Q_0 = 1 then perform A ← A – B and set Q[0] = 0.

Flowchart

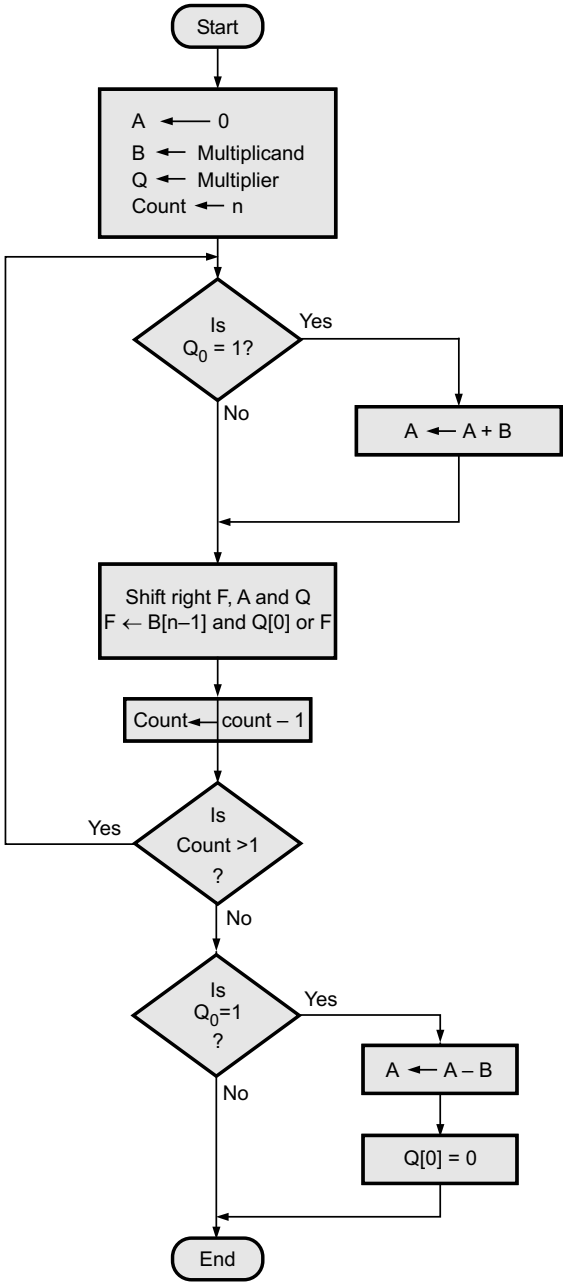


Fig. 2.6.4

Let us see the multiplication of 4-bit fractions, 0101 and 0100 with all possible combinations.

Here,

$$\text{Multiplicand} = 0101 = -2^0 \times 0 + 2^{-1} \times 1 + 2^{-2} \times 0 + 2^{-3} \times 1$$

$$= 0 + 0.5 + 0 + 0.125 = 0.625$$

$$= 1011 = -2^0 \times 1 + 2^{-1} \times 0 + 2^{-2} \times 1 + 2^{-3} \times 1$$

$$= -1.000 + 0 + 0.250 + 0.125 = -0.625$$

$$\text{Multiplier} = 0100 = -2^0 \times 0 + 2^{-1} \times 1 + 2^{-2} \times 0 + 2^{-3} \times 0$$

$$= 0 + 0.5 + 0 + 0 = 0.5$$

$$= 1100 = -2^0 \times 1 + 2^{-1} \times 1 + 2^{-2} \times 0 + 2^{-3} \times 0$$

$$= -1.000 + 0.5 + 0 + 0 = -0.5$$

CASE 1 : Both positive (0.625 × 0.5)

Multiplicand (B) ← 0 1 0 1 (0.625) Multiplier (Q) ← 0 1 0 0 (0.5)				
Steps	**F**	**A**	**Q**	**Operation**
	0	0 0 0 0	0 1 0 0	Initial
Step 1	0	0 0 0 0	0 0 1 0	Shift right
Step 2	0	0 0 0 0	0 0 0 1	Shift right
Step 3	0	0 1 0 1	0 0 0 1	A ← A + B (A ← 0 0 0 0 + 0 1 0 1)
	0	0 0 1 0	1 0 0 0	Shift right
Step 4 (Final)	0	0 0 1 0	1 0 0 0	Nothing
Result :		0 0 1 0	1 0 0 0 = 0.3125	

CASE 2 : Negative multiplier (0.625 × – 0.5)

Multiplicand (B) ← 0 1 0 1 (0.625) Multiplier (Q) ← 1 1 0 0 (– 0.5)				
Steps	**F**	**A**	**Q**	**Operation**
	0	0 0 0 0	1 1 0 0	Initial
Step 1	0	0 0 0 0	0 1 1 0	Shift right
Step 2	0	0 0 0 0	0 0 1 1	Shift right
Step 3	0	0 1 0 1	0 0 1 1	A ← A + B (A ← 0 0 0 0 + 0 1 0 1)
	0	0 0 1 0	1 0 0 1	Shift right
Step 4	0	1 1 0 1	1 0 0 1	A ← A – B (A ← 0 0 1 0 + 1 0 1 1)
		1 1 0 1	1 0 0 0	Set Q [0] to 0
Result :		1 1 0 1	1 0 0 0 = – 0.3125	

CASE 3 : Negative Multiplicand (– 0.625 × 0.5)

Multiplicand (B) ← 1 0 1 1 (– 0.625) Multiplier (Q) ← 0 1 0 0 (0.5)				
Steps	**F**	**A**	**Q**	**Operation**
	0	0 0 0 0	0 1 0 0	Initial
Step 1	0	0 0 0 0	0 0 1 0	Shift right
Step 2	0	0 0 0 0	0 0 0 1	Shift right
Step 3	1	1 0 1 1	0 0 0 1	A ← A + B (A ← 0 0 0 0 + 1 0 1 1)
	1	1 1 0 1	1 0 0 0	Shift right
Step 4	1	1 1 0 1	1 0 0 0	Nothing
Result :		1 1 0 1	1 0 0 0 = – 0.3125	

CASE 4 : Both Negative (– 0.625 × – 0.5)

Multiplicand (B) ← 1 0 1 1 (– 0.625) Multiplier (Q) ← 1 1 0 0 (– 0.5)				
Steps	**F**	**A**	**Q**	**Operation**
	0	0 0 0 0	1 1 0 0	Initial
Step 1	0	0 0 0 0	0 1 1 0	Shift right
Step 2	0	0 0 0 0	0 0 1 1	Shift right
Step 3	1	1 0 1 1	0 0 1 1	A ← A + B (A ← 0 0 0 0 + 1 0 1 1)
	1	1 1 0 1	1 0 0 1	Shift right
Step 4	1	0 0 1 0	1 0 0 1	A ← A – B (A ← 1 1 0 1 + 0 1 0 1)
		0 0 1 0	1 0 0 0	Set Q [0] to 0
Result :		0 0 1 0	1 0 0 0 = 0.3125	

Review Questions

1. *Explain the Robertson algorithm for 2's complement multiplication .*
2. *Explain the Robertson algorithm for 2's complement fractions.*
3. *Draw the block diagram of an n-bit 2's complement adder-subtractor.* **Dec.-03, Marks 2**
4. *Explain 2's complement multiplier with a neat block diagram.* **Dec.-11, Marks 16**

2.7 Booth's Algorithm

Dec.-03, 08, 09, 10, 12, May-06, 07, 08, 09, 11

In this section we discuss multiplication of 2's complement signed operands. Here also we follow the same strategy that to accumulate partial products by adding versions of multiplicand (either positive or negative) as selected by the multiplier bit. It is important to note that multiplicand when we add negative multiplicand we must extend the sign-bit value of the multiplicand to the left as far as product will extend. Refer Fig. 2.7.1. The sign extension of the multiplicand is shown bold. We now discuss a technique which works equally well for both negative and positive multiplier, called **Booth algorithm**.

A powerful algorithm for signed-number multiplication is a Booth's algorithm, which generates a 2n-bit product and treats both positive and negative numbers uniformly.

This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers. For example, multiplier 0 0 1 1 1 0 (14) can be represented as follows.

```
  0 1 0 0 0 0 (16)
– 0 0 0 0 1 0 (2)
 ----------------------
  0 0 1 1 1 0 (14)
```

Therefore, the product can be computed by adding 2^4 times the multiplicand to the 2's complement of 2^1 times the multiplicand. In simple notations, we can describe the sequence of required operations by **recoding** the preceding multiplier as

0 + 1 0 0 – 1 0

In general, for Booth's algorithm recoding scheme can be given as :

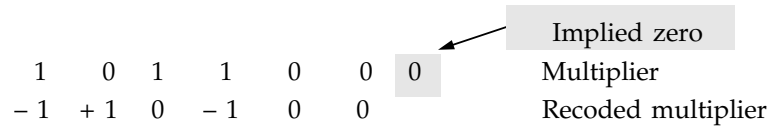
*–1 times the shifted multiplicand is selected when moving from 0 to 1, +1 times the shifted multiplicand is selected when moving from 1 to 0, and 0 times the shifted multiplicand is selected for none of the above case, as multiplier is scanned from **right to left**.*

We have to assume an implied 0 to right of the multiplier LSB. This is illustrated in the following examples.

Example 2.7.1 *Recode the multiplier 1 0 1 1 0 0 for Booth's multiplication.*

Dec.-09, Marks 2

Solution :



Example 2.7.2 *Recode the multiplier 0 1 1 0 0 1 for Booth's multiplication.*

Solution :

						Implied zero	
	0	1	1	0	0	1	0	Multiplier
+	1	0	–1	0	+1	–1		Recoded multiplier

The Fig. 2.7.1 shows the Booth's multiplication. As shown in the Fig. 2.7.1, whenever multiplicand is multiplied by –1, its 2's complement is taken as a partial result.

Multiplier : 0 0 1 1 0 0 Multiplicand : 0 1 0 0 1 1.

Recoded multiplier : 0 + 1 0 – 1 0 0

						0	1	0	0	1	1	**Multiplicand**
					×	0	+1	0	–1	0	0	**Multiplier**
	0	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	
+	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0		
+	**1**	**1**	**1**	**1**	1	0	1	1	0	1			← **2's complement of the multiplicand**
+	**0**	**0**	**0**	0	0	0	0	0	0				
+	**0**	**0**	0	1	0	0	1	1					
+	**0**	0	0	0	0	0	0						
	0	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	

Fig. 2.7.1 Booth's multiplication

The same algorithm can be used for negative multiplier. This is illustrated in the following example.

Example 2.7.3 *Multiply 0 1 1 1 0 (+14) and 1 1 0 1 1 (–5).*

Solution :

0	1	1	1	0	(+14)	**Multiplicand**
1	1	0	1	1	(–5)	**Multiplier**
0	**–1**	**1**	**0**	**–1**		**Recoded multiplier**

Multiplication :

						0	1	1	1	0	**Multiplicand**
					×	0	–1	+1	0	–1	**Recoded multiplier**
	1	**1**	**1**	**1**	**1**	1	0	0	1	0	← **2's complement of the multiplicand**
+	**0**	**0**	**0**	**0**	0	0	0	0	0		
+	**0**	**0**	**0**	0	1	1	1	0			
+	**1**	**1**	1	0	0	1	0				
+	**0**	0	0	0	0	0					
	1	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**(–70)**

The same algorithm also can be used for negative multiplier and negative multiplicand. This is illustrated in the following example.

Example 2.7.4 *Explain the following pair of signed 2's complement numbers.*

Multiplicand : 1 1 0 0 1 1 (–13)

Multiplier : 1 0 1 1 0 0 (–20)

Solution :

1	0	1	1	0	0	**Multiplier**
–1	+1	0	–1	0	0	**Recoded multiplier**

Multiplication :

						1	1	0	0	1	1	**Multiplicand**
					×	–1	+1	0	–1	0	0	**Recoded multiplier**
	0	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	
+	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0		
+	**0**	**0**	**0**	**0**	0	0	1	1	0	1			← **2's complement of the multiplicand**
+	**0**	**0**	**0**	0	0	0	0	0	0				
+	**1**	**1**	1	1	0	0	1	1					
+	**0**	0	0	1	1	0	1						← **2's complement of the multiplicand**
	0	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**(260)**

The Booth's algorithm can be implemented as shown in the Fig. 2.7.2. The circuit is similar to circuit for positive number multiplication. It consists of n-bit adder, shift, add subtract control logic and four registers, A, B, Q and Q_{-1}. As shown in the Fig. 2.7.2 multiplier and multiplicand are loaded into register Q and register B, respectively, and register A and Q_{-1} are initially set to 0. The sequence counter, SC is set to a number n equal to the number of bits in the multiplier.

The n-bit adder performs addition of two inputs. One input is the A register and other input is multiplicand. In case of addition, $\overline{\text{Add}}$/sub line is 0, therefore C_{in}= 0 and multiplicand is directly applied as a second input to the n-bit adder. In case of subtraction, $\overline{\text{Add}}$/sub line is 1, therefore C_{in}=1 and multiplicand is complemented and then applied to the n-bit adder. As a result, the 2's complement of multiplicand is added in the A register.

The shift, add and subtract control logic scans bits Q_0 and Q_{-1} one at a time and generates the control signals as shown in the Table 2.7.1. If the two bits are same (1 – 1 or 0 – 0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to right 1-bit without addition or subtraction (Add/subtract Enable = 0). If the two bits are differ, then the multiplicand (B-register) is added to or subtracted from the A register, depending on the status of bits. If bits are Q_0 = 0 and Q_{-1} = 1 then multiplicand is

added and if bits are $Q_0 = 1$ and $Q_{-1} = 0$ then multiplicand is subtracted. After addition or subtraction right shift occurs such that the leftmost bit of A (A_{n-1}) is not only shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an *arithmetic shift*, since it preserves the sign bit. After arithmetic shift, the sequence counter is decremented by 1 and if it is not zero, computational loop is repeated. That is, in all computational loop is repeated n-times.

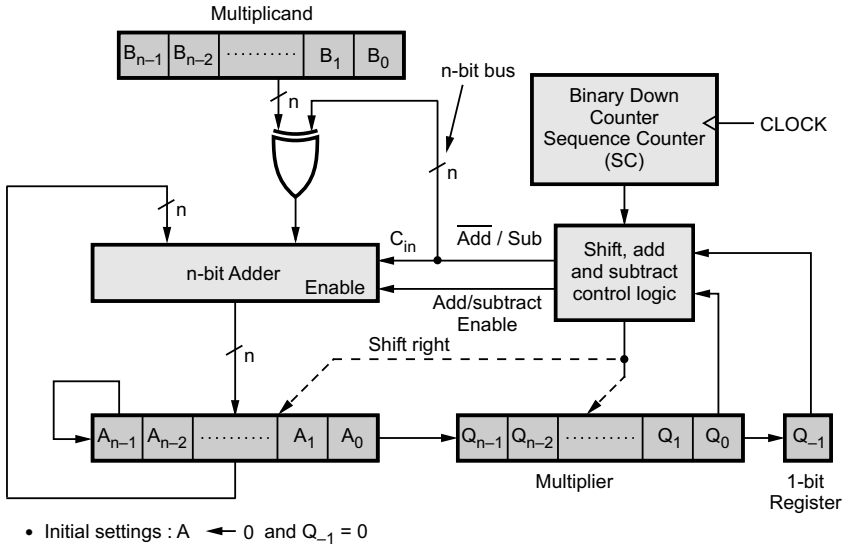


Fig. 2.7.2 Hardware implementation of signed binary multiplication

Q_0	Q_{-1}	$\overline{Add}$/Sub	Add/Subtract Enable	Shift
0	0	×	0	1
0	1	0	1	1
1	0	1	1	1
1	1	×	0	1

Table 2.7.1 Truth table for shift, add and subtract control logic

The sequence of events in Booth's algorithm can be explained with the help of flowchart shown in Fig. 2.7.3.

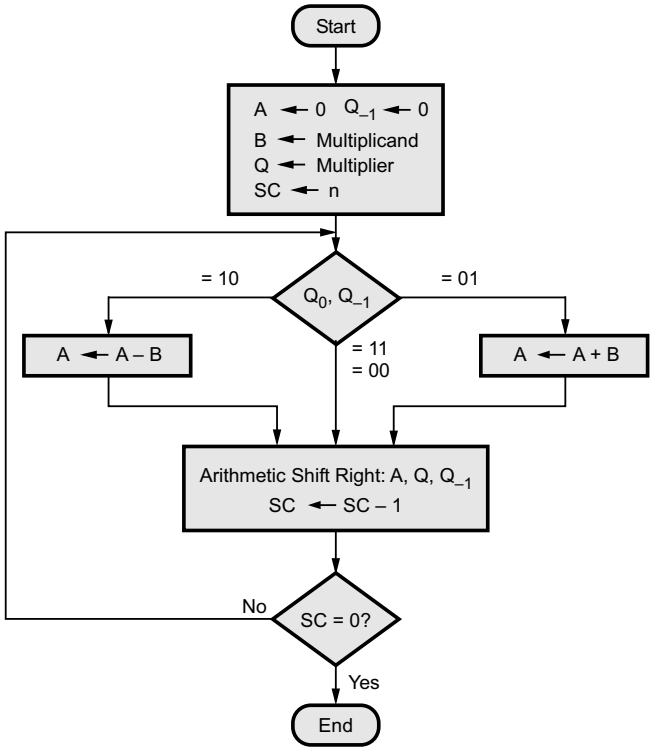


Fig. 2.7.3 Booth's algorithm for signed multiplication

Let us see the multiplication of 4-bit numbers, 5 and 4 with all possible combinations.

Case 1 : Both Positive (5×4)

Multiplicand (B) = 0 1 0 1 (5) and Multiplier (Q) = 0 1 0 0 (4)													
SC				A				Q					Operation
				A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1	0	0		0	0	0	0	0	1	0	**0**	**0**	Initial
0	1	1		0	0	0	0	0	0	1	**0**	**0**	Arithmetic shift right
0	1	0		0	0	0	0	0	0	0	**1**	**0**	Arithmetic shift right
				0	0	0	0						**$Q_0 Q_{-1}$ = 1 0**
			+	1	0	1	1						
				1	**0**	**1**	**1**	0	0	0	1	0	∴ A ← A – B
0	0	1		1	1	0	1	1	0	0	**0**	**1**	Arithmetic shift right
				1	1	0	1						**$Q_0 Q_{-1}$ = 0 1**
			+	0	1	0	1						
				0	**0**	**1**	**0**	1	0	0	0	1	∴ A ← A + B
0	0	0		0	0	0	1	0	1	0	**0**	**0**	Arithmetic shift right
Result : =				**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**		**= + $(20)_{10}$**

Case 2 : Negative Multiplier (5 × – 4)

Multiplicand (B) = 0 1 0 1 (5) and Multiplier (Q) = 1 1 0 0 (–4)													
SC				A				Q					Operation
				A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1	0	0		0	0	0	0	1	1	0	**0**	**0**	Initial
0	1	1		0	0	0	0	0	1	1	**0**	**0**	Arithmetic shift right
0	1	0		0	0	0	0	0	0	1	**1**	**0**	Arithmetic shift right
				0	0	0	0						**$Q_0 Q_{-1}$ = 1 0**
			+	1	0	1	1						
				1	**0**	**1**	**1**	0	0	1	1	0	∴ A ← A – B
0	0	1		1	1	0	1	1	0	0	**1**	**1**	Arithmetic shift right
0	0	0		1	1	1	0	1	1	0	**0**	**1**	Arithmetic shift right
Result : =				**1**	**1**	**1**	**0**	**1**	**1**	**0**	**0**		**= – 20 (2's complement of 20)**

Case 3 : Negative Multiplicand (– 5 × 4)

Multiplicand (B) = 1 0 1 1 (–5) and Multiplier (Q) = 0 1 0 0 (4)													
SC				A				Q					Operation
				A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1	0	0		0	0	0	0	0	1	0	**0**	**0**	Initial
0	1	1		0	0	0	0	0	0	1	**0**	**0**	Arithmetic shift right
0	1	0		0	0	0	0	0	0	0	**1**	**0**	Arithmetic shift right
				0	0	0	0						**$Q_0 Q_{-1}$ = 1 0**
			+	0	1	0	1						
				0	**1**	**0**	**1**	0	0	0	1	0	∴ A ← A – B
0	0	1		0	0	1	0	1	0	0	**0**	**1**	Arithmetic shift right
				0	0	1	0						**$Q_0 Q_{-1}$ = 0 1**
			+	1	0	1	1						
				1	**1**	**0**	**1**	1	0	0	0	1	∴ A ← A + B
0	0	0		1	1	1	0	1	1	0	**0**	**0**	Arithmetic shift right
Result : =				**1**	**1**	**1**	**0**	**1**	**1**	**0**	**0**		**= – 20 (2's complement of 20)**

Case 4 : Both Negative (– 5 × – 4)

Multiplicand (B) = 1 0 1 1 (–5) and Multiplier (Q) = 1 1 0 0 (4)													
SC				A				Q					Operation
				A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1	0	0		0	0	0	0	1	1	0	**0**	**0**	Initial
0	1	1		0	0	0	0	0	1	1	**0**	**0**	Arithmetic shift right
0	1	0		0	0	0	0	0	0	1	**1**	**0**	Arithmetic shift right
				0	0	0	0						$Q_0 Q_{-1}$ = 1 0
			+	0	1	0	1						
				0	**1**	**0**	**1**	0	0	1	1	0	∴ A ← A – B
0	0	1		0	0	1	0	1	0	0	**1**	**1**	Arithmetic shift right
0	0	0		0	0	0	1	0	1	0	**0**	**1**	Arithmetic shift right
Result : =				**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**		**= + 20**

Example 2.7.5 *Explain the Booth's algorithm for multiplication of signed 2's complement numbers. Using it show the step-by-step multiplication* $(-9)\times(-13)$.

Dec.-03, 16 Marks

Solution :

Multiplicand (B) = 1 0 1 1 1 (–9) and Multiplier (Q) = 1 0 0 1 1 (–13)														
SC				A					Q						Operation
				A_4	A_3	A_2	A_1	A_0	Q_4	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1	0	1		0	0	0	0	0	1	0	0	1	**1**	**0**	Initial
				0	0	0	0	0							$Q_0 Q_{-1}$ = 1 0
			+	0	1	0	0	1							
				0	**1**	**0**	**0**	**1**	1	0	0	1	1	0	∴ A ← A – B
1	0	0		0	0	1	0	0	1	1	0	0	**1**	**1**	Arithmetic shift right
0	1	1		0	0	0	1	0	0	1	1	0	**0**	**1**	Arithmetic shift right
				0	0	0	1	0							$Q_0 Q_{-1}$ = 0 1
			+	1	0	1	1	1							
				1	**1**	**0**	**0**	**1**	0	1	1	0	0	1	∴ A ← A + B
0	1	0		1	1	1	0	0	1	0	1	1	**0**	**0**	Arithmetic shift right
0	0	1		1	1	1	1	0	0	1	0	1	**1**	**0**	Arithmetic shift right
				1	1	1	1	0							$Q_0 Q_{-1}$ = 1 0
			+	0	1	0	0	1							
				0	**0**	**1**	**1**	**1**	0	1	0	1	1	0	∴ A ← A – B
0	0	0		0	0	0	1	1	1	0	1	0	**1**	**1**	Arithmetic shift right
Result : =				**0**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**		**= + 117**

Review Questions

1. *Discuss the principle behind the Booth's multiplier.* **May-07, 08, Marks 2**
2. *Illustrate Booth's algorithm with an example.* **May-08, Marks 16**
3. *Design a multiplier that multiplies two 4-bit numbers.* **May-07, Marks 8**
4. *Give the algorithm for multiplication of signed 2's complement numbers and explain its operation.* **May-08, Marks 8**
5. *Multiply (– 12) and (13) using Booth's algorithm.* **Dec.-08, Marks 8**
6. *Design the full adder circuit and explain how it is used in the multiplications of two signed number.* **May-09, Marks 16**
7. *Describe in detail booth's multiplication algorithms and its hardware implementation.* **Dec.-10, Marks 16**
8. *Give the Booth's algorithm for two's complement multiplication and illustrate the algorithm with an example.* **May-11, Dec.-12, Marks 6**

2.8 Fast Multiplication

May-06, 07, 08, 09, 10, 11, Dec.-07, 08

There are two techniques for speeding up the multiplication process. In first technique the maximum number of summands are reduced to n/2 for n-bit operands. The second technique, called the carry save addition reduces the time needed to add the summand. In this section we will see first technique to speed up multiplication.

2.8.1 Bit-Pair Recoding of Multipliers

To speed-up the multiplication process in the Booth's algorithm a technique called **bit-pair recoding** is used. It is also called **modified Booth's algorithm**. It halves the maximum number of summands.

In this technique, the Booth-recoded multiplier bits are grouped in pairs. Then each pair is represented by its equivalent single bit multiplier reducing total number of multiplier bits to half. For example pair (+ 1 –1) is equivalent to the pair (0 +1). That is, instead of adding –1 times multiplicand at shifted position i to +1 times the multiplicand at position i + 1, the same result is obtained by adding +1 times multiplicand at position i. Similarly, (+1 0) is equivalent to (0 +2), (–1 +1) is equivalent to (0 –1) and so on.

By replacing pairs with their equivalents we can get bit-pair recoded multiplier. But instead of deriving bit-pair recoded multiplier from Booth recoded multiplier one can directly derive it from original multiplier. The bit-pair recoding of multiplier can be directly derived from Table 2.8.1 The Table 2.8.1 shows the bit-pair code for all possible multiplier bit options.

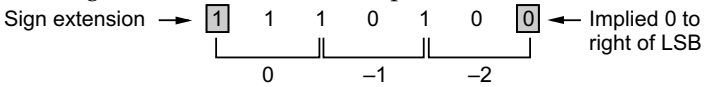
Multiplier bit-pair		Multiplier bit on the right	Bit-pair recoded multiplier bit at position i
i + 1	i	i – 1	
0	0	0	0
0	0	1	+ 1
0	1	0	+ 1
0	1	1	+ 2
1	0	0	– 2
1	0	1	– 1
1	1	0	– 1
1	1	1	0

Table 2.8.1

Example 2.8.1 *Find the bit-pair code for multiplier.*

1 1 0 1 0.

Solution : By referring table we can derive bit-pair code as follows :

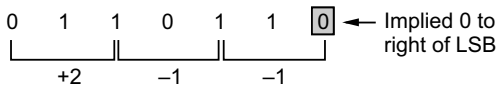


Example 2.8.2 *Multiply given signed 2's complement numbers using bit-pair recoding*

A = 1 1 0 1 0 1 multiplicand (–11)

B = 0 1 1 0 1 1 multiplier (+27)

Solution : Let us find the bit-pair code for multiplier.



Multiplication :

						1	1	0	1	0	1		**Multiplicand**
					×		+2		–1		–1		**Multiplier**
	0	**0**	**0**	**0**	**0**	**0**	0	0	1	0	1	1	← **2's complement of the multiplicand**
+	**0**	**0**	**0**	**0**	0	0	1	0	1	1			← **2's complement of the multiplicand**
+	**1**	**1**	1	0	1	0	1	0					← **Multiplicand × (+2)**
	1	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**(–297)**

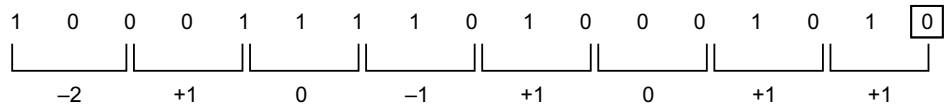
Example 2.8.3 *Give the Booth's recoding and bit-pair recoding of the number.*

1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 1. **May-06, Marks 2**

Solution : Booth's recoding

1	0	0	0	1	1	1	1	0	1	0	0	0	1	0	1	0
–1	0	0	+1	0	0	0	–1	+1	–1	0	0	+1	–1	+1	–1	

Bit-Pair Recoding

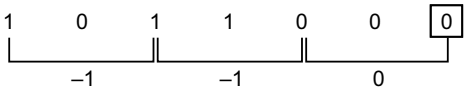


Example 2.8.4 *Multiply the following pair of signed 2's complement numbers using bit-pair recoding of the multipliers : A = 010111, B = 101100.* **May-06, Marks 4**

Solution : A = 0 1 0 1 1 1 Multiplicand (+ 23)

B = 1 0 1 1 0 0 Multiplier (– 20)

Let us find the bit-pair code for multiplier



Multiplication

					0	1	0	1	1	1	
				×			–1	–1	0		
	0	**0**	**0**	**0**	0	0	0	0	0	0	
+	**1**	**1**	1	0	1	0	0	1			**2's complement of the multiplicand**
+	**1**	0	1	0	0	1					**2's complement of the multiplicand**
	1	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**(–460)**

Example 2.8.5 *Explain Booth's algorithm to multiply the following pair of signed two's complement numbers :*

A = 110011 multiplicand B = 101100 multiplier

Also, implement the above using bit-pair Recoding and explain how it achieves faster multiplication. **May-07, Marks 8**

Solution :

Multiplicand (B) ← 110011 (–13) Multiplier (Q) ← 101100 (–20)				
Steps	**A**	**Q**	**Q – 1**	**Operation**
	0 0 0 0 0 0	1 0 1 1 0 0	0	Initial
1.	0 0 0 0 0 0	0 1 0 1 1 0	0	Shift right
2.	0 0 0 0 0 0	0 0 1 0 1 1	0	Shift right
3.	0 0 1 1 0 1	0 0 1 0 1 1	0	A ← A ← B
	0 0 0 1 1 0	1 0 0 1 0 1	1	Shift right
4.	0 0 0 0 1 1	0 1 0 0 1 0	1	Shift right

5.	1 1 0 1 1 0	0 1 0 0 1 0	1	A ← A + B
	1 1 1 0 1 1	0 0 1 0 0 1	0	Shift right
6.	0 0 1 0 0 0	0 0 1 0 0 1	0	A ← A – B
	0 0 0 1 0 0	0 0 0 1 0 0	1	Shift right
Result :	0 0 0 1 0 0	0 0 0 1 0 0	= + 260	

Implementation with Bit-Pair Recoding

Bit-pair Recoding for Multiplier

Multiplication

					×	1	1	0	0	1	1	
								–1	–1	0		
	0	**0**	**0**	**0**	**0**	0	0	0	0	0	0	**Multiplicand × 0**
+	**0**	**0**	**0**	0	0	1	1	0	1			**2's complement of the multiplicand**
+	**0**	0	0	1	1	0	1					**2's complement of the multiplicand**
	0	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**(+260)**

The Booth's algorithm may need the summation at each step and number of steps required in Booth's algorithm are equal to length of multiplier in bits. The bit-pair recoding halves the maximum number of summations. Hence it achieves faster multiplication.

Example For Practice

Example 2.8.6 : *Illustrate multiplication of signed 2's complement numbers 01101 and 11010 using bit-pairing of the multipliers.* **May-09, Marks 8**

2.8.2 Array Multiplier

The multiplication process for binary numbers is similar to the decimal numbers. Actually binary multiplication is simple than decimal multiplication since it involves only 1s and 0s.

Rules for binary multiplication

$0 \times 0 = 0$ $1 \times 0 = 0$

$0 \times 1 = 0$ $1 \times 1 = 1$

Example 2.8.7 *Multiply 011_2 by 110_2 using binary multiplication method.*

Solution :

We have seen how multiplication takes place in binary numbers. It uses n shifts and adds to multiply n-bit binary number. The combination logic circuit implemented to perform such multiplication is called combinational multiplier or **array multiplier.**

		0	1	1	
	×	1	1	0	
		0	**0**	**0**	
+	0	1	1	0	← **Shift left**
+	0	1	1	0	0 ← **Shift left**
	1	**0**	**0**	**1**	**0**

Let us generalize the multiplication process for a 2×2 multiplier for two unsigned 2-bit numbers : multiplicand $A = A_1\ A_0$ and multiplier $B = B_1\ B_0$. The Fig. 2.8.1 shows how the multiplication process is carried out.

The multiplication process involves multiplication (product) of 2-bit number and addition of 2-bit number. The multiplication of 2-bits can be implemented using 2-input AND gate whereas addition of 2-bits can be implemented using half-adder. Such an implementation of 2×2 multiplier is shown in the Fig. 2.8.2.

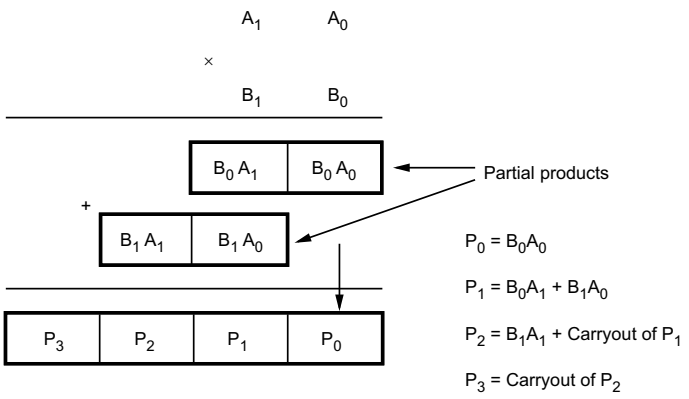


Fig. 2.8.1 Multiplication process

The Fig. 2.8.2 shows the multiplication process for a 4×4 multiplier for two unsigned integers: multiplicand $A = A_3\ A_2\ A_1\ A_0$ and multiplier $B = B_3\ B_2\ B_1\ B_0$. As shown in Fig. 2.8.3, each shifted multiplicand which is multiplied by either 0 or 1 depending on the corresponding multiplier bit is called partial product. Each partial product consists of four product component. The product is represented by a product term surrounded by rectangular box. The final 8-bit product is obtained by adding all partial products.

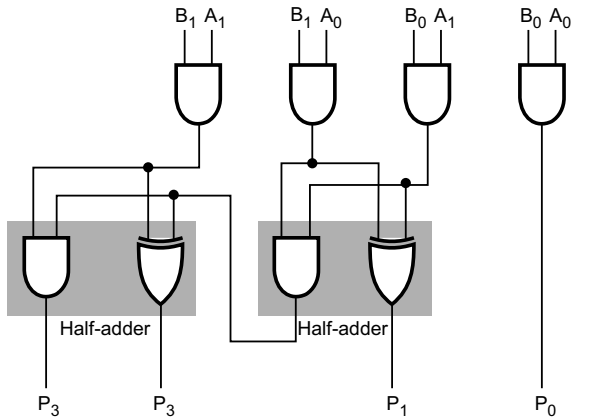


Fig. 2.8.2 2 × 2 bit combinational array multiplier

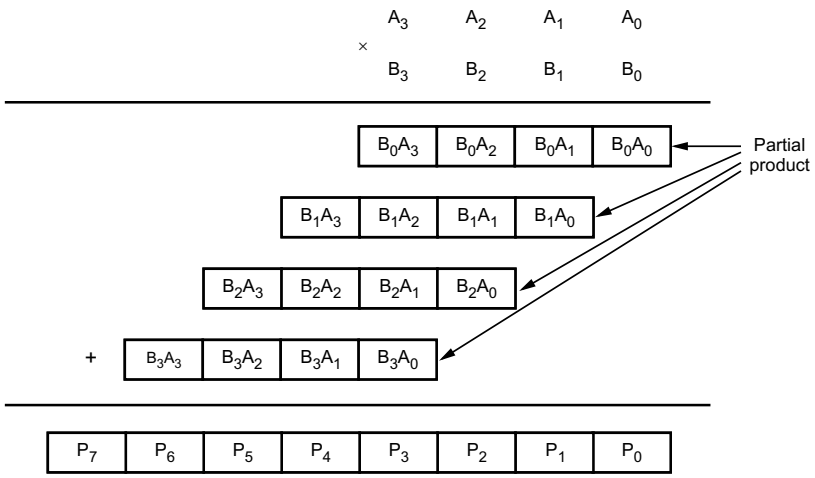


Fig. 2.8.3 4 bit by 4-bit binary multiplication

The first partial product is formed by multiplying B_0 by A_3 A_2 A_1 A_0. The second partial product is formed by multiplying B_1 by A_3 A_2 A_1 A_0. The third partial is formed by multiplying B_2 by A_3 A_2 A_1 A_0 and the fourth partial product is formed by multiplying B_3 by A_3 A_2 A_1 A_0.

The multiplication of two bits such as B_0 and A_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to AND operation. Therefore, the partial products can be implemented with AND gates as shown in Fig. 2.8.4.

The 4-bit partial products are added using 4-bit parallel adder. During addition of first partial product, three most significant bits of it are added to the second partial product. As we take only three bits from first partial product, the fourth (most significant bit) is considered as 0. This is illustrated in Fig. 2.8.4.

The three most significant bits and carry out (treated as most significant bit) of first partial sum are then added to the third partial product. Finally the three most significant bits and carry out (treated as most significant bit) of second partial sum are added to the fourth partial product.

The carryout and third sum represents the five most significant bits of the product.

Least significant bits of first and second partial sum represents P_1 and P_2, respectively, and product B_0A_0 represents P_0.

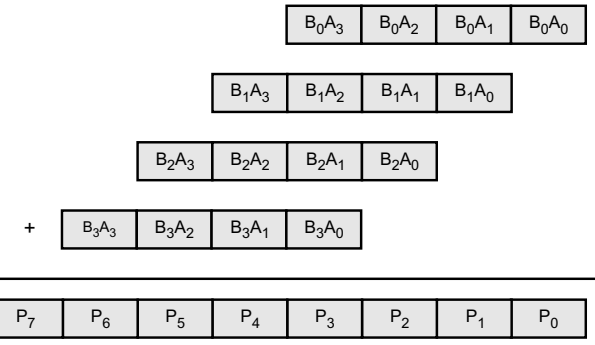


Fig. 2.8.5 Multiplication process of 4 × 4 multiplier

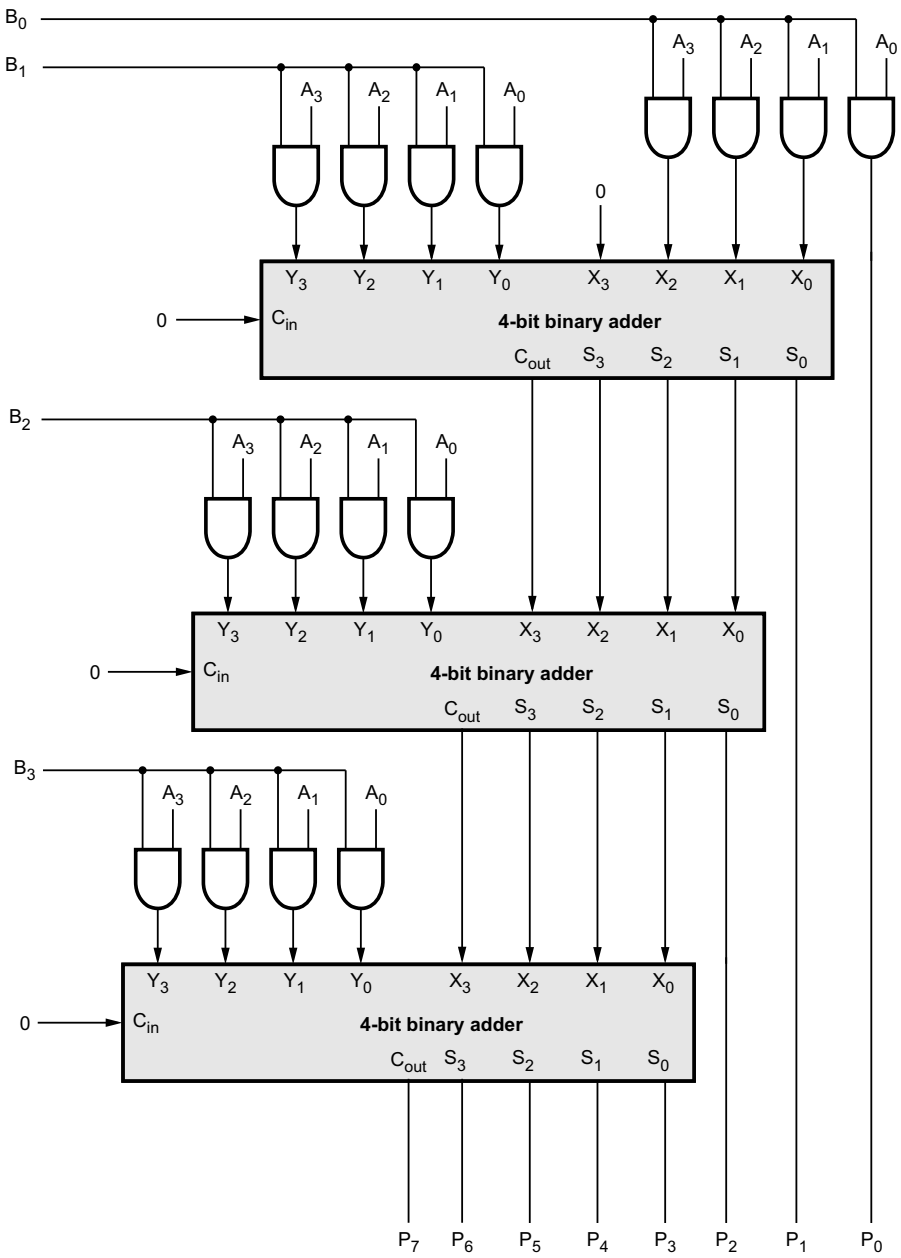


Fig. 2.8.4 4-bit by 4-bit binary multiplier

The 4-bit by 4-bit binary multiplier circuit discussed above can be drawn using full adders as shown in the Fig. 2.8.5, each shifted multiplicand which is multiplied by either

0 or 1 depending on the corresponding multiplier bit is called partial product. Each partial product consists of four product component. The product component bit is a logical AND of multiplier bit B_i and multiplicand bit A_j, i.e. $B_i \times A_j$ and it is represented by a product term surrounded by rectangular box. The final 8-bit product is obtained by adding all partial products.

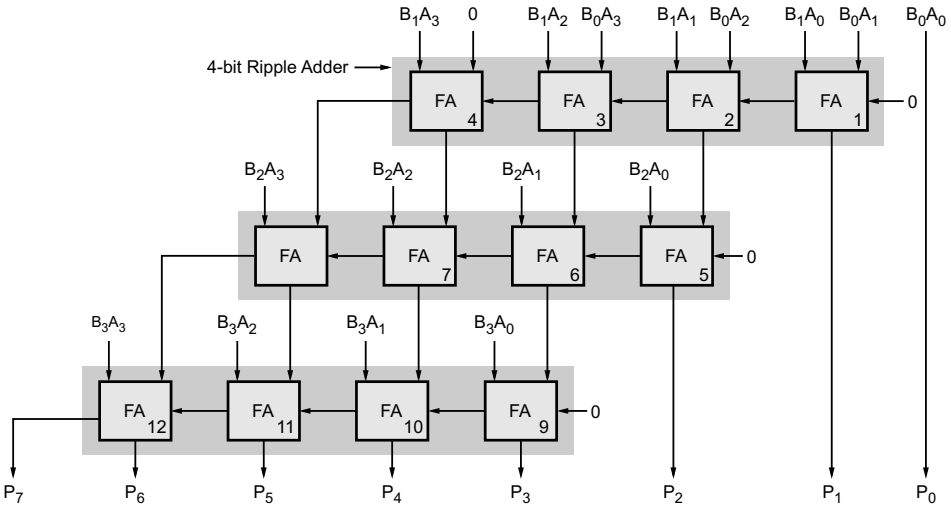


Fig. 2.8.6 4 × 4 combinational multiplier

The Fig. 2.8.6 shows the circuit to add the product components. Here, the product components are separated to make the space, and each `+' box is a full adder. The carries in each partial product row of full adders are connected to make an 4-bit ripple adder. Thus, the first 4-bit ripple adder adds the first two rows of product components to produce the first partial product. The carry output generated is propagated to the most significant product component used to produce the next partial product. The subsequent adders adds each partial product with the next product component.

In the above multiplier, the least significant adder is full-adder 1 and the adder which gives the MSB of the product (P_7) is full-adder 12. If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst-case path goes through 8 full-adders (1, 2, 3, 4, 7, 8, 11 and 12) and its delay is 8 t_{pd}.

To increase the speed of the addition process many times technique called **carry-save addition** is used. In this technique, the carry output from bit i during step j is applied to carry input for bit i + 1 during the next step, j + 1. After addition of product components in the last row, one more step is required in which the carries are allowed to ripple from the least to the most significant bit. The Fig. 2.8.7 shows 4 × 4 combinational multiplier using carry-save addition technique. In this the carry out of

each full adder in the first seven row of full adders are connected to an input of an adder below it. Carries in the fourth row of full adders are connected to create a conventional ripple adder. This technique does not save any hardware but it reduces the propagation delay substantially.

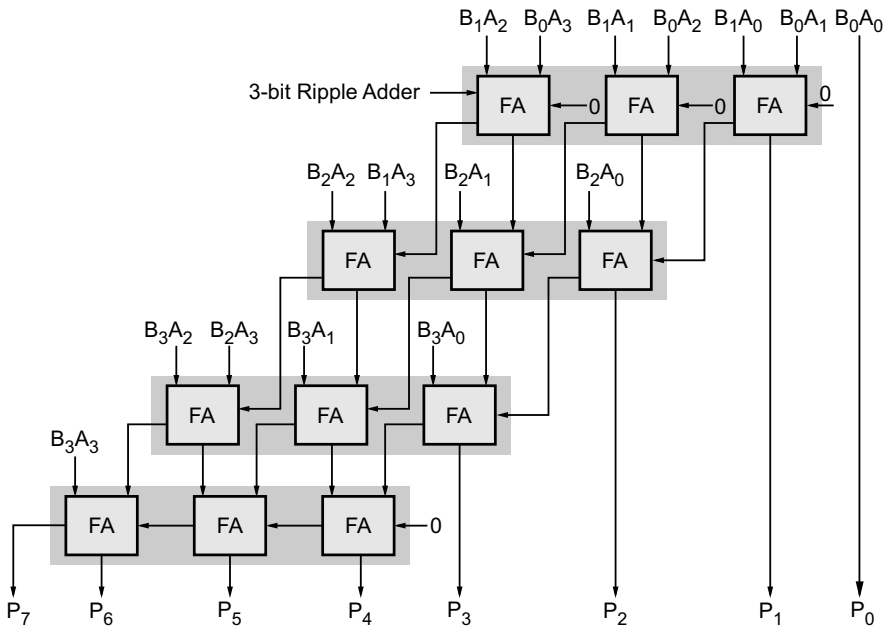


Fig. 2.8.7 4 × 4 combinational multiplier with carry save addition

Review Questions

1. *Discuss the role of Booth's algorithm in the design of fast multipliers.* **May-08, Marks 2**
2. *Explain the modified Booth's algorithm.*
3. *Design an array multiplier that multipliers two 4 bit numbers. Use AND gates and binary address.*
4. *Draw and explain 2-bit by 2-bit array multiplier.*
5. *Discuss the concept of bit pair recoding.*
   Multiply the following pair of signed 2's complement numbers using bit-pair recoded multiplier : Multiplicand = 110011 and Multiplier = 101100. **May-07, Marks 8**
6. *Write about the CSA method of fast multiplication. Prove how it is faster with an example.* **Dec.-07, Marks 8**
7. *Design an array multiplier that multiplies two 4-bit numbers and explain its operation.* **May-08, Marks 8**

8. *Discuss the role of Booth's algorithm in the design of fast multipliers.* **May-08, Marks 2**
9. *Explain the ripple carry multiplier with $(m_3m_2m_1m_0)(q_3q_2q_1q_0)$ inputs.* **Dec.-08, Marks 8**
10. *Illustrate multiplication of signed 2's complement numbers 01101 and 11010 using bit-pairing of the multipliers.* **May-09, Marks 8**
11. *Draw the logical circuit for array multiplication.* **May-10, Marks 2**
12. *With suitable diagram explain how multiplication can be performed using a multistage carry-save adder circuit and also the implementation of a pipelined carry-save multiplier.* **May-11, Marks 10**

2.9 Division

Dec.-07, 08, 09, May-06, 07, 09, 11, 12

The division is more complex than multiplication. For simplicity we will see division for positive numbers. Fig. 2.9.1 shows the usual algorithm for dividing positive numbers by hand. It shows examples of decimal division and the binary-coded division of the same value.

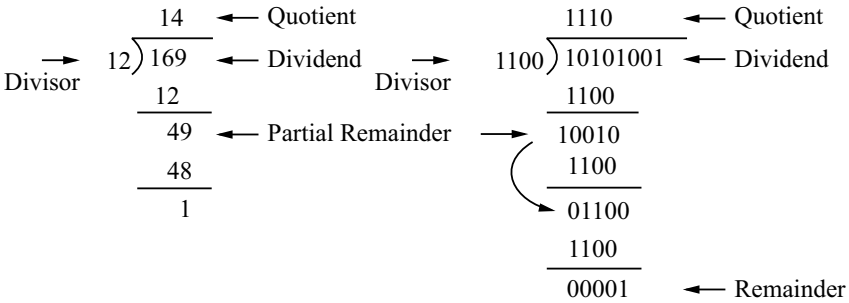


Fig. 2.9.1 Division examples

In both the divisions, division process is same, only in binary division quotient bits are 0 and 1. We will now see the binary division process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this condition occurs, 0s are placed in the quotient from left to right. When the condition is satisfied, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a **partial remainder**. From this point onwards, the division process follows repetition of steps. In each repetition cycle, additional bits from the dividend are brought down to the partial remainder until the result is greater than or equal to the divisor, and the divisor is subtracted from the result to produce a new partial remainder. The process continues until all the bits of the dividend are brought down and result is still less than the divisor.

2.9.1 Restoring Division

Fig. 2.9.2 shows the hardware for implementation of division process described in the previous section. It consists of n + 1-bit binary adder, shift, add and subtract control logic and registers A, B, and Q. As shown in the Fig. 2.9.2 divisor and dividend are loaded into register B and register Q, respectively. Register A is initially set to zero. The division operation is then carried out. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A.

Division Operation Steps :

1. Shift A and Q left one binary position.
2. Subtract divisor from A and place answer back in A ($A \leftarrow A - B$).
3. If the sign bit of A is 1, set Q_0 to 0 and add divisor back to A (that is, restore A); Otherwise, set Q_0 to 1.
4. Repeat steps 1, 2, and 3 n times.

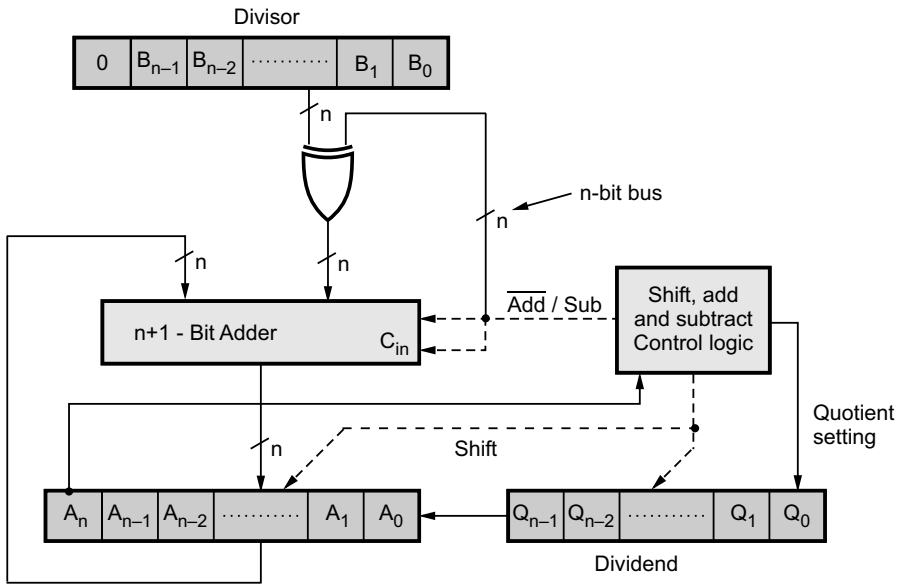


Fig. 2.9.2 Hardware to implement binary division

A flowchart for division operation is as shown in Fig. 2.9.3.

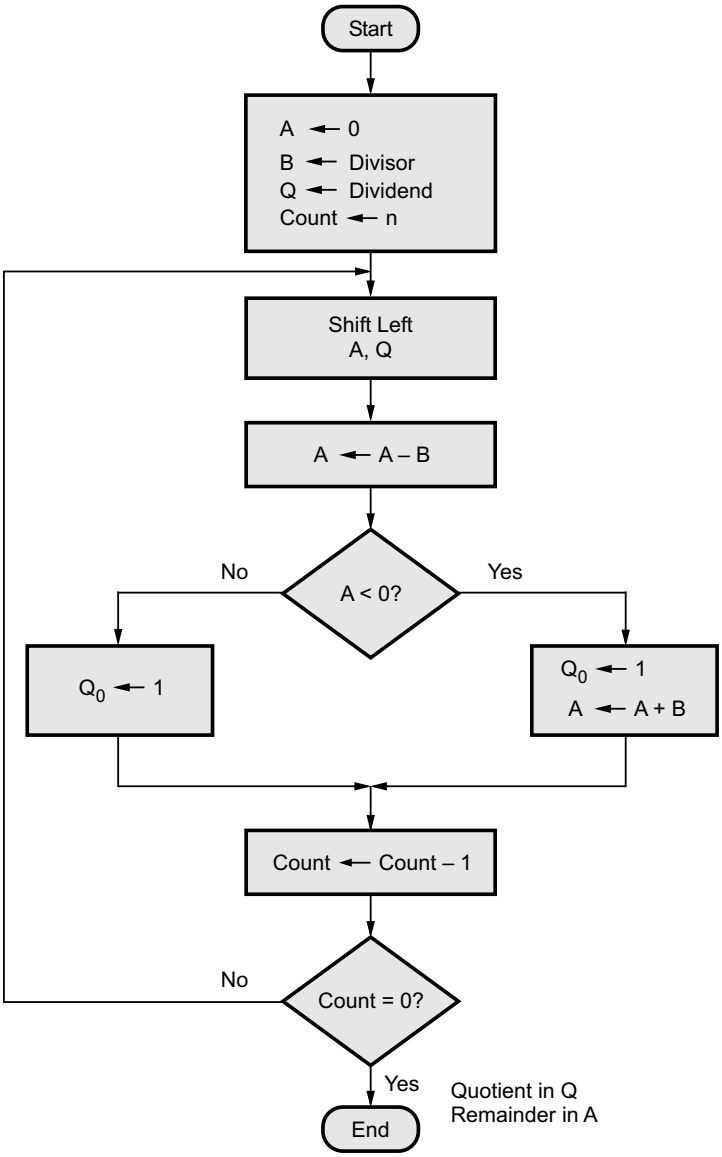


Fig. 2.9.3 Flowchart for restoring division operation

Let us see one example. Consider 4-bit dividend and 2-bit divisor :

Dividend = 1 0 1 0

Divisor = 0 0 1 1

Fig. 2.9.4 shows steps involved in the above binary division.

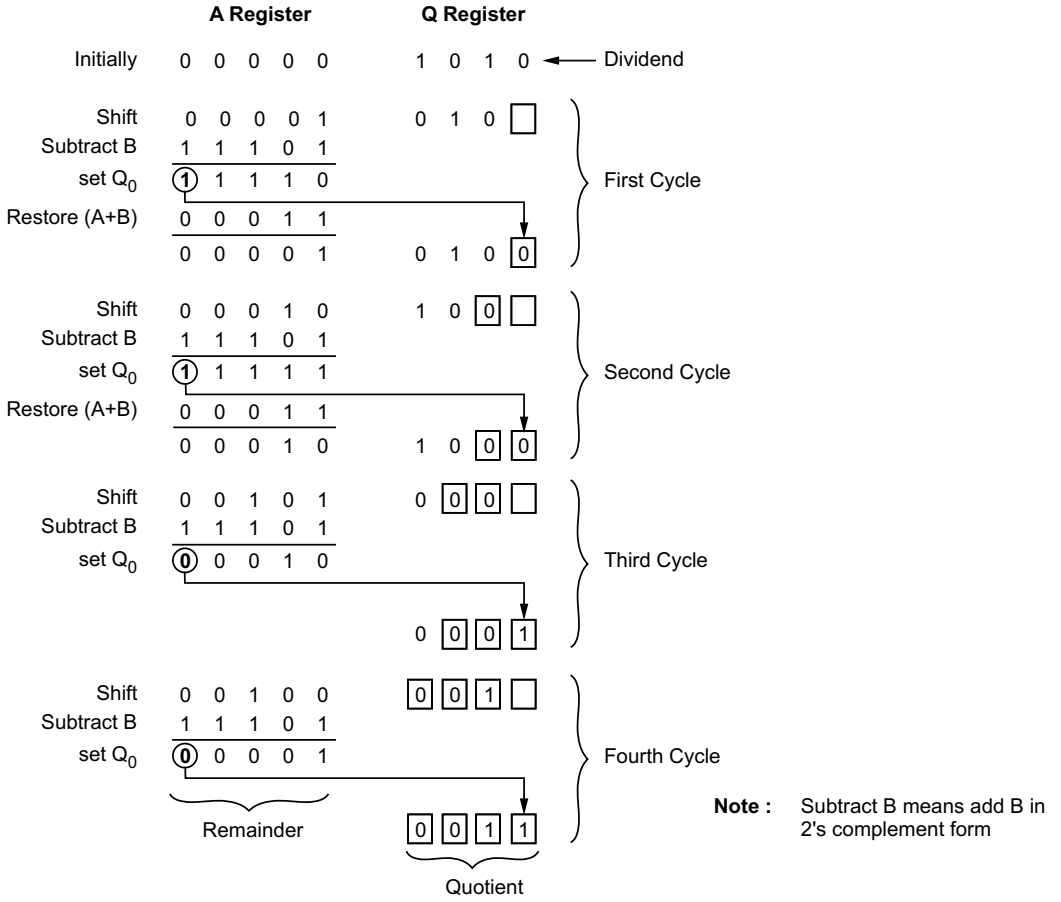


Fig. 2.9.4 A restoring division example

The division algorithm just discussed needs restoring register A after each unsuccessful subtraction. (Subtraction is said to be unsuccessful if the result is negative). Therefore it is referred to as **restoring division algorithm**. This algorithm is improved, giving *non-restoring division* algorithm. Consider the sequence of operations that takes place after the subtraction operation in the **restoring algorithm**.

If A is positive	If A is negative
Shift left and subtract divisor $\rightarrow$ 2A – B	Restore $\rightarrow$ A + B Shift left and subtract divisor$\rightarrow$ 2 (A + B) – B $\equiv$ 2A + B

Looking at the above operations we can write following steps for non-restoring algorithm.

Step 1 : If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A; otherwise, shift A and Q left and add divisor to A. If the sign of A is 0, set Q_0 to 1; otherwise, set Q_0 to 0.

Step 2 : Repeat steps 1 and 2 for n times.

Step 3 : If the sign of A is 1, add divisor to A.

Note : Step 3 is required to leave the proper positive remainder in A at the end of n cycles.

Example 2.9.1 *Perform 1100 ÷ 11 using restoring algorithm .*

Solution : Restoring Division : Q ← Dividend (1100), B ← Divisor (11).

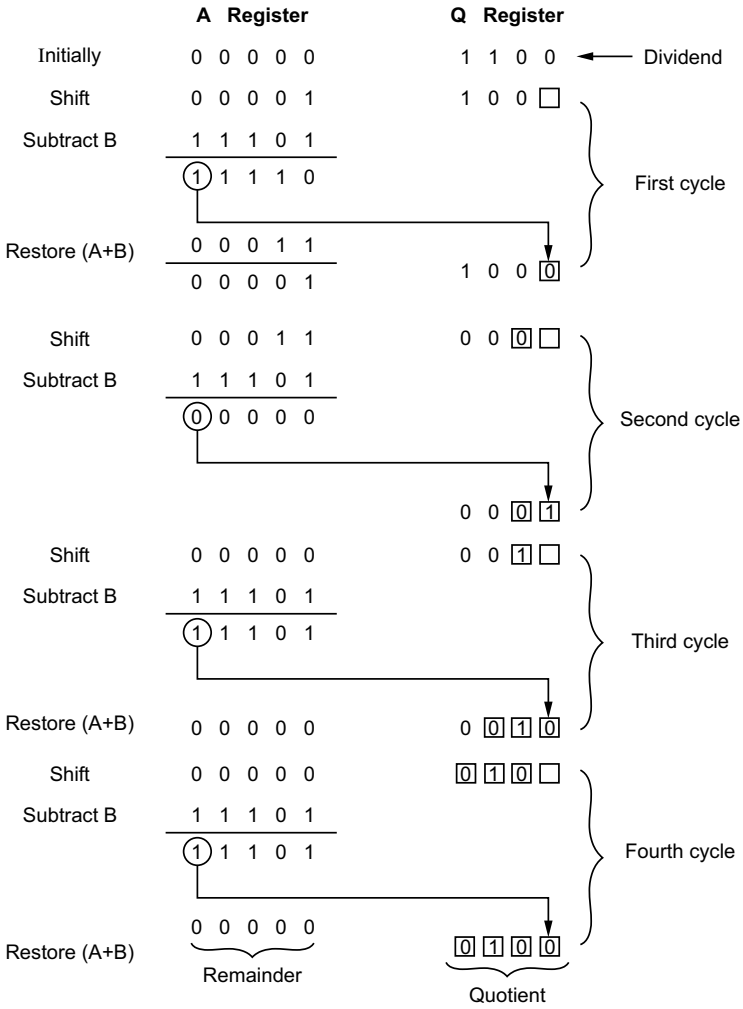


Fig. 2.9.5 (a)

Example for Practice

Example 2.9.2 : *Demonstrate the division of 1100_2 by 101_1 using restoring method, draw block diagram and explain the operation.* **Dec.-08, Marks 16**

2.9.2 Non-restoring Division

A flowchart for non-restoring division operation is as shown in Fig. 2.9.6. Let us see one example. Consider 4-bit dividend and 2-bit divisor :

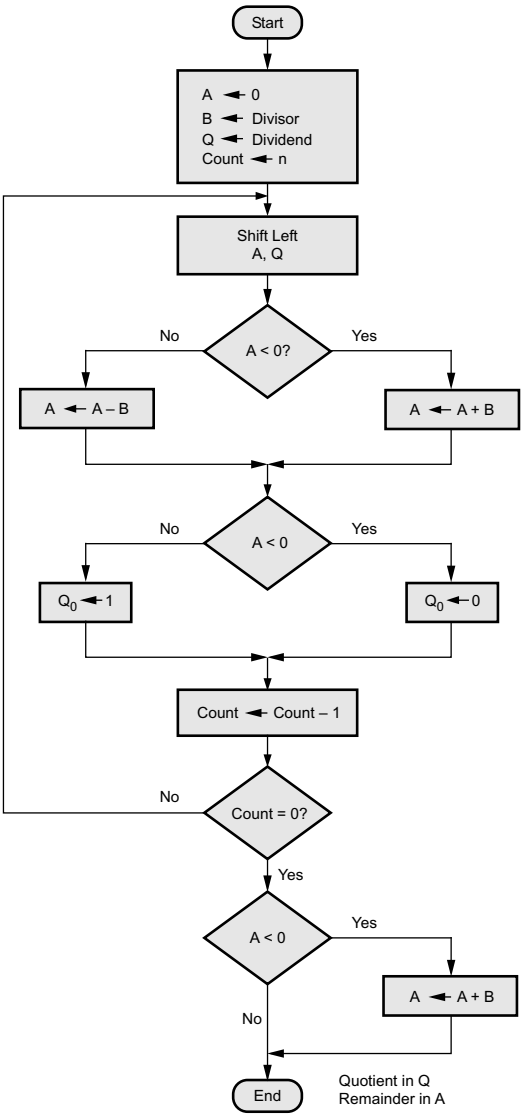


Fig. 2.9.6 Flowchart for non-restoring division operation

Dividend = 1 0 1 0

Divisor = 0 0 1 1

Fig. 2.9.7 shows steps involved in the non-restoring binary division.

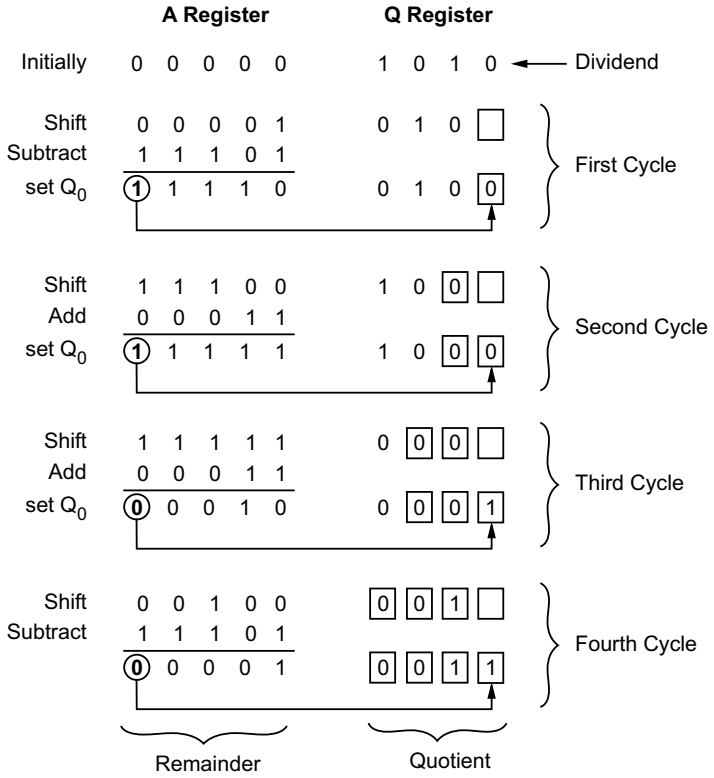


Fig. 2.9.7 A non-restoring division example

In the above example after 4 cycles register A is positive and hence step 3 is not required. Let us see another example where we need step 3. Consider 4-bit dividend and 2-bit divisor :

Dividend = 1 0 1 1

Divisor = 0 1 0 1

Fig. 2.9.8 shows steps involved in the non-restoring binary division.

The hardware shown in Fig. 2.9.2 can also be used to perform non-restoring algorithm. There is no simple algorithm for signed division. In signed division, the operands are preprocessed to transform them into positive values. Then using one of the algorithms just discussed quotients and remainders are calculated. The quotients and remainders are then transformed to the correct signed values.

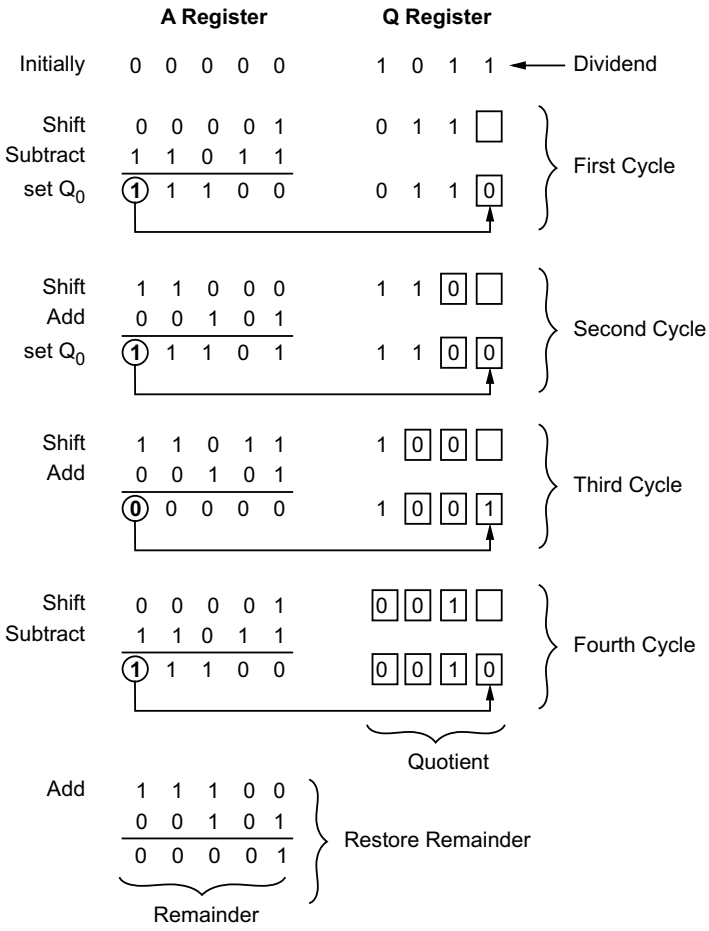


Fig. 2.9.8 A non-restoring division example

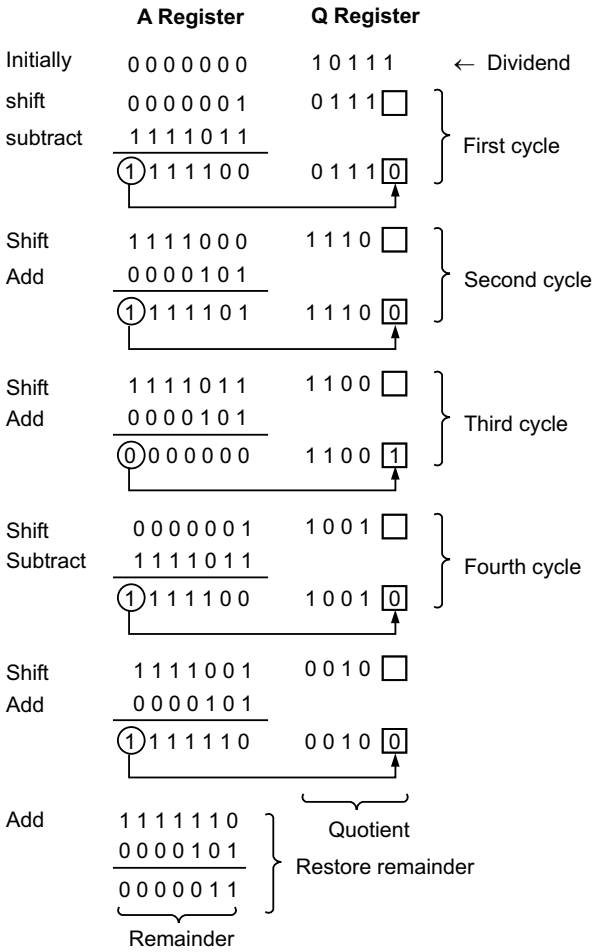
Example 2.9.3 *Using the non-restoring division algorithm, compute 23/5.*

May-07, Marks 8

Solution :

$$\text{Dividend} = 23_{10} = 10111$$

$$\text{Divisor} = 5_{10} = 00101$$



Example 2.9.4 *Perform 1100 ÷ 11 using non-restoring algorithm*

Solution : Non-Restoring Division : Q ← Dividend (1100), B ← Divisor (11).

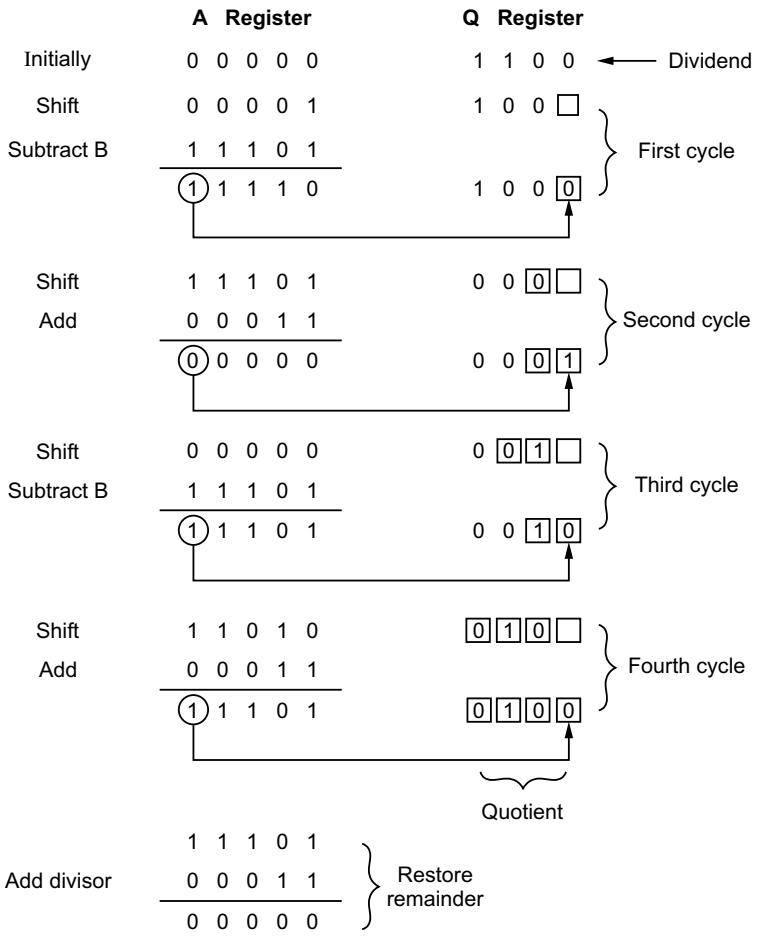


Fig. 2.9.9

Examples for Practice

Example 2.9.5 : *Explain non-restoring division algorithm with the help of suitable example.* **May-09, Marks 8**

Example 2.9.6 : *Perform the division on the following 5-bit unsigned integer using non-restoring division : 10101/00101.* **May-06, Marks 4**

Review Question

1. *Divide 21 (twenty one) by 3 (three) using non-restoring method and explain the steps involved, with the neat diagram.* **May-12, Marks 16**

2.9.3 Comparison between Restoring and Non-restoring Division Algorithm

Sr. No.	Restoring	Non-restoring
1.	Needs restoring of register A if the result of subtraction is negative.	Does not need restoring.
2.	In each cycle content of register A is first shifted left and then divisor is subtracted from it.	In each cycle content of register A is first shifted left and then divisor is added or subtracted with the content of register A depending on the sign of A.
3.	Does not need restoring of remainder.	Needs restoring of remainder if remainder is negative.
4.	Slower algorithm.	Faster algorithm.

Review Questions

1. *Describe the algorithm for integer division with suitable example.* **May-07, Dec.-07, 09, Marks 8**
2. *Explain the restoring division algorithm.* **May-09, Marks 8**
3. *Explain non-restoring division algorithm with the help of suitable example.* **May-09, Marks 8**
4. *Give the comparison between restoring and non-restoring division.*
5. *Derive non-restoring division algorithm for unsigned integers and illustrate with an example.* **May-11, Marks 8**

2.10 Floating Point Arithmetic

Dec.-06, 07, 08, 12, May-06, 09, 10

In this section we are going to see general procedures for addition, subtraction, multiplication and division of floating point numbers.

2.10.1 Addition and Subtraction

Consider two floating point numbers :

$$X = m_1 \cdot r^{e1} \text{ and}$$

$$Y = m_2 \cdot r^{e2} \text{ Assume : } e_1 \geq e_2$$

The rules for addition and subtraction and are :

Step 1 : Select the number with a smaller exponent and shift its mantissa right, a number of steps equal to the difference in exponents $|e_2 - e_1|$. For examples, if the numbers are 1.75×10^2 and 6.8×10^4, then the number 1.75×10^2 is selected and converted to 0.0175×10^4

Step 2 : Set the exponent of the result equal to the larger exponent.

Step 3 : Perform addition/subtraction on the mantissas and determine the sign of the result.

Step 4 : Normalize the result, if necessary.

Let us perform the addition and subtraction of single precision numbers using above rules.

Example 2.10.1 *Add single precision floating point numbers A and B, where A = 44900000H and B = 42A00000H*

Solution : Step 1 : Represent numbers in single precision format

A = 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 … 0

B = 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 … 0

Exponent for A = 1 0 0 0 1 0 0 1 = 137

∴ Actual exponent = 137 – 127 (Bias) = 10

Exponent for B = 1 0 0 0 0 1 0 1 = 133

∴ Actual exponent = 133 – 127 (Bias) = 6

∴ Number B has smaller exponent with difference 4. Hence its mantissa is shifted right by 4-bits as shown below.

Step 2 : Shift mantissa

Shifted mantissa of B = 0 0 0 0 0 1 0 0 0

Step 3 : Add mantissas

Mantissa of A = 0 0 1 0 0 0 0 0 0

Mantissa of B = 0 0 0 0 0 1 0 0 …. 0

Mantissa of result = 0 0 1 0 0 1 0 0 …. 0

As both numbers are positive, sign of the result is positive.

∴ Result = 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 … 0

= 44920000 H

We have seen addition process for floating point numbers. Similar process is used for subtraction of floating point numbers. In subtraction, two mantissas are subtracted instead of addition and the sign of greater mantissa is assigned to the result.

Example 2.10.2 *Subtract single point precision floating point numbers A and B (A – B), where A = 4 4 9 0 0 0 0 0 H and B = 4 2 A 0 0 0 0 0 H.*

Solution : Step 1 : Represent numbers in single precision format.

A = 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 ….0

B = 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0

Exponent for A = 1 0 0 0 1 0 0 1 = 137

∴ Actual exponent = 137 – 127 (Bias) = 10

Exponent for B = 1 0 0 0 0 1 0 1 = 133

∴ Actual exponent = 133 – 127 (Bias) = 6

∴ Number B has smaller exponent with difference 4. Hence its mantissa is shifted right by 4 bits as shown below :

Step 2 : Shift mantissa.

Shifted mantissa of B = 0 0 0 0 0 1 0 0 … 0

Step 3 : Subtract mantissa.

Mantissa of A = 0 0 1 0 0 0 0 0 0

Mantissa of B = 0 0 0 0 0 1 0 0 0

0 0 0 1 1 1 0 0 0

Mantissa for A is greater than mantissa for B therefore sign of result is sign of A.

∴ Result = 0 1 0 0 0 1 0 0 1 0 0 0 1 1 1 0 0 … 0

= 4 4 8 E 0 0 0 0 H

2.10.2 Problems in Floating Point Arithmetic

Mantissa overflow : The addition of two mantissas of the same sign may result in a carryout of the most significant bit. If so, the mantissa is shifted right and the exponent is incremented.

Mantissa underflow : In the process of aligning mantissas, digits may flow off the right end of the mantissa. In such case truncation methods such as chopping, rounding are used.

Exponent overflow : Exponent overflow occurs when a positive exponent exceeds the maximum possible exponent value. In some systems this may be designated as $+\infty$ or $-\infty$.

Exponent underflow : Exponent underflow occurs when a negative exponent exceeds the maximum possible exponent value. In such cases, the number is designated as zero.

2.10.3 Flowchart and Algorithm for Floating Point Addition and Subtraction

Fig. 2.10.1 shows flowchart for floating point addition and subtraction.

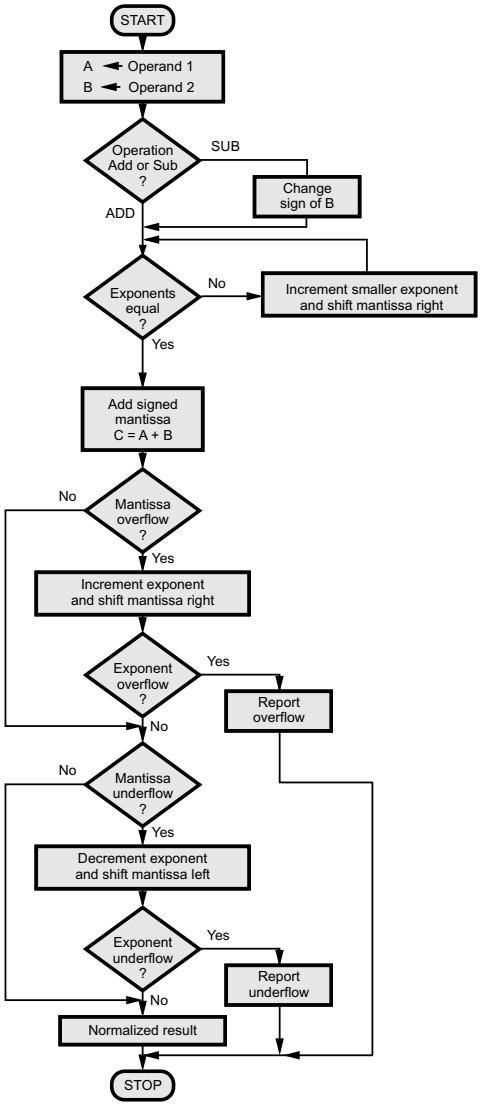


Fig. 2.10.1 Flowchart for floating point addition and subtraction

Algorithm : Referring the flowchart given in Fig. 2.10.1 it is possible to write algorithm for floating point addition as follows.

Declare registers : AM [0 : M – 1] ; For mantissa of A
BM [0 : M – 1] ; For mantissa of B
AE [0 : E – 1] ; For exponent of A
BE [0 : E – 1] ; For exponent of B
E [0 : E – 1] ; For temporary storage of exponent

Step 1 : Read numbers : AM ← Mantissa of A
BM ← Mantissa of B
AE ← Exponent of A
BE ← Exponent of B

Step 2 : Compare the two exponents : E = AE – BE

Step 3 : Equalize

if E < 0 then
right shift AM ;
E = E + 1 and go to step 3 ;

if E > 0 then
right shift BM ;
E = E – 1 go to step 3 ;

Step 4 : Add mantissas : AM = AM + BM and E = Max (AE, BE)

Step 5 : Check exponent overflow :

if overflow occurs during mantissa addition then
if E = Emax then report overflow
else
right shift AM
E = E + 1 go to stop

Step 6 : Adjust for zero result

if AM = 0 then E = 0 ; go to stop

Step 7 : Normalize result

if AM normalized then go to stop

Step 8 : Check exponent underflow

if E > Emin then
left shift AM ;
E = E – 1 go to step 7

Step 9 : Stop

2.10.4 Implementing Floating-Point Operations

The Fig. 2.10.2 shows the hardware implementation for the addition and subtraction of 32-bit floating point operands that have the single precision format, i.e. 1-bit for sign, 8-bits for signed exponent and 23-bits for mantissa. As per the rules of

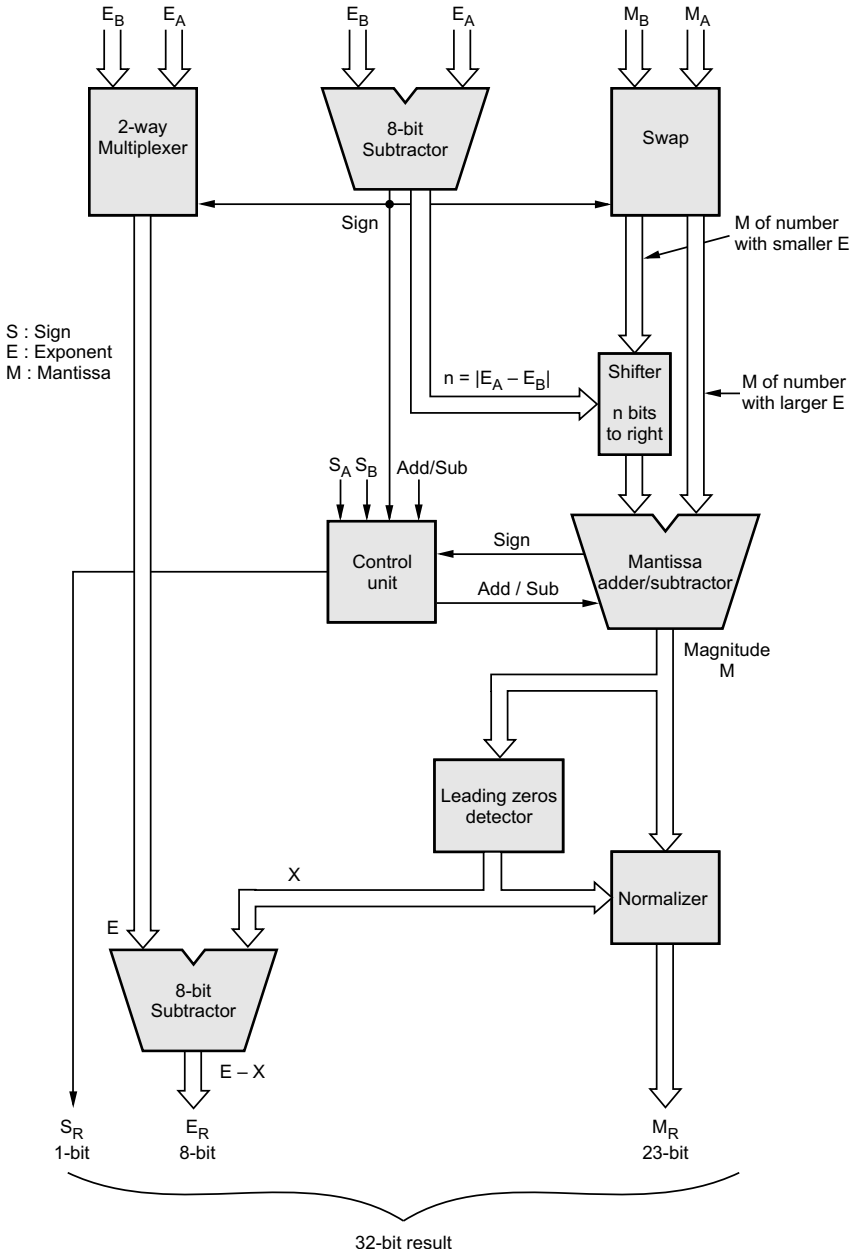


Fig. 2.10.2

addition/subtraction of floating point number we have to first compare the exponents of numbers to determine a number with a smaller exponent and then to determine how far to shift the mantissa of that number so that its exponent matches with the other number.

The shift count value, n, is determined by the 8-bit subtractor. The inputs for the 8-bit subtractor are exponent values of two number and the operation performed is $E_A - E_B$. The subtraction gives the difference, n. This difference, is sent to the SHIFTER unit.

The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. If the sign is 0, then $E_A \geq E_B$ and input to SWAP network is 0. This disables swapping and mantissa M_B is sent to the SHIFTER. If the sign is 1, the $E_A - E_B$ and input to SWAP network is 1. In this case swapping is enabled and mantissa M_A is sent to the SHIFTER. The SHIFTER unit shifts the given mantissa n positions to the right.

The two way multiplexer is used to set the exponent of the result (E) equal to the larger exponent, i.e. step 5. Multiplexer has two inputs E_A and E_B and its output is based on the sign of the difference resulting from comparing exponents in step 1. The output of multiplexer is

$$E = E_A \text{ if } E_A \geq E_B$$

or $$E = E_B \quad E_A < E_B$$

The third step is to perform addition or subtraction on the mantissas and determine the sign of the result. The control logic is used to determine whether the mantissas are to be added or subtracted. It decides this by checking the signs of the operands (S_A and S_B) and the operation (Add or Subtract) that is to be performed on the operands. The control logic is also responsible for determining the sign of the result (S_R). The control logic determines the sign of the result by checking the resulted sign of mantissa adder/subtractor, sign from the exponent comparison, signs of the operands and the operation to be performed.

In step 4, the result of the mantissa (M) is normalized. The normalized value is truncated to generate the 23-bit mantissa, MR, of the result. The leading zeros detector determines the number of bit shifts, X, to be applied to mantissa (M). The value X is then subtracted from the tentative resulted exponent E to generate the true result exponent, E_R.

2.10.5 Multiplication and Division

In floating point arithmetic, multiplication and division are somewhat easier than addition and subtraction because an alignment of mantissas is not required in multiplication and division.

The rules of multiplication are :

1. Add the exponents and subtract bias. (127 in case of single precision numbers and 1023 in case of double precision numbers).

2. Multiply the mantissas and determine the sign of the result.
3. Normalize the result.

Fig. 2.10.3 shows the flowchart for floating point multiplication.

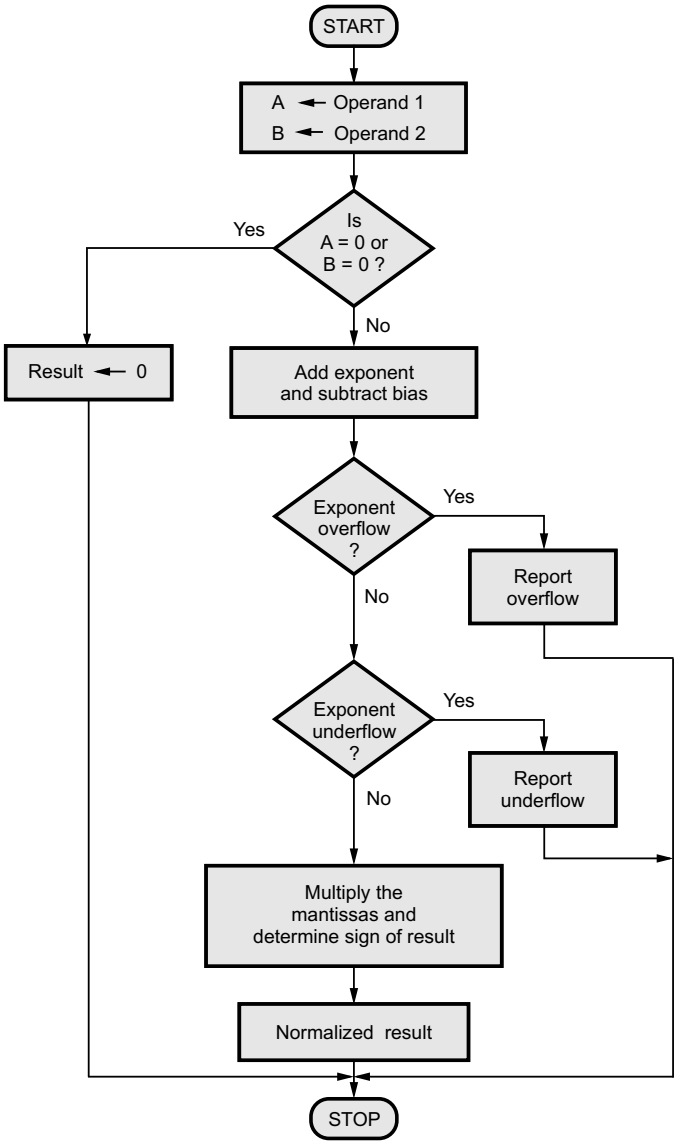


Fig. 2.10.3 Flowchart for floating point multiplication

The rules of division are :

1. Subtract exponents and add bias (127 in case of single precision numbers and 1023 in case of double precision numbers).

2. Divide the mantissas and determine the sign of the result.
3. Normalize the result.

Fig. 2.10.4 shows the flowchart for floating point division.

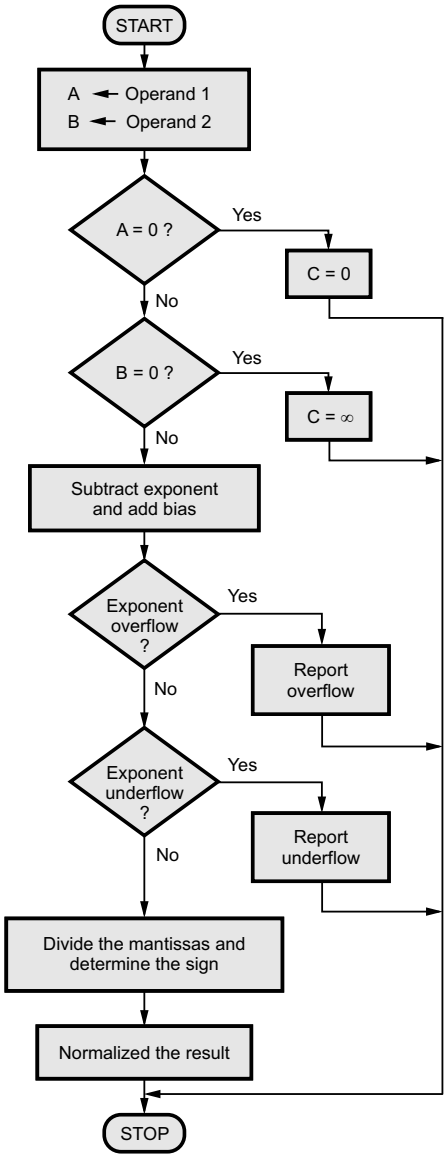


Fig. 2.10.4 Flowchart for floating point division

2.10.6 Guard Bits and Truncation

We know that the mantissas of initial operands and final results are limited to 24-bits, including the implicit leading 1. But if we provide extra bits in the intermediate steps of calculations we can get maximum accuracy in the final result. These extra bits used in the intermediate calculations are called **guard bits**. Thus, operands are used with guard bits for calculations and then final result of calculation is truncated to create 24-bit number that approximates the longer final result.

There are three commonly used methods of truncation :

- Chopping
- Von Neumann rounding
- Rounding

Chopping : This is the simplest method of truncation. Here, the guard bits are removed without making any changes in the retained bits.

For example, if we want to truncate a fraction from six to three bits by this method we have to remove 3 least significant bits (right most). This is illustrated in Fig. 2.10.5.

Original Number	$0.\ b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}$
Truncated Number	$0.\ b_{-1}\ b_{-2}\ b_{-3}$

Fig. 2.10.5 Chopping Method of truncation

In our example, the error in the 3-bit result ranges from 0 to .000111. The value .000111 is almost equal to .001. Because of this in general terms we can say that, the error in chopping ranges from 0 to almost 1 in the least significant position of the retained bits. In our example, least significant position of the retained bit is b_{-3}. In chopping method, the error range is not symmetrical about 0 and hence the result of chopping is a **biased approximation**.

Von Neumann Rounding : This is the next simplest method of truncation. Here,

- If the bits to be removed are all 0s, they are simply removed, with no changes to the retained bits.
- However, if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1.

In our 6-bit to 3-bit truncation we get results as shown in the Fig. 2.10.6.

Original number	$0.\ b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}$ with $b_{-4}\ b_{-5}\ b_{-6} = 0\ 0\ 0$	$0.\ b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}$ with $b_{-4}\ b_{-5}\ b_{-6} \neq 0\ 0\ 0$
Truncated number	$0.\ b_{-1}\ b_{-2}\ b_{-3}$	$0.\ b_{-1}\ b_{-2}\ 1$

Fig. 2.10.6 Von Neumann method of truncation

The error in Von Neumann method ranges between – 1 and + 1 in the LSB position of the retained bits. In this method the range of error is larger than it is with chopping; however, the maximum error magnitude is the same. In this method, the error range is symmetrical about 0 (Refer Fig. 2.10.7) and hence the result of Von Neumann rounding is a **unbiased approximation**. Unbiased approximations are advantageous if many operands and operations are involved in generating a result, because positive errors tend to offset negative errors as computation proceeds.

Original Number	0.010010	0.011010
Truncated Number	0.011	0.011
	Positive Error	Negative Error

Fig. 2.10.7

Rounding : It is the best method of truncation; however it is the most difficult to implement. In this method, a 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed. In our 6-bit to 3-bit example we will get results as shown in Fig. 2.10.8.

Original number	$0.\ b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}$ with $b_{-4} = 0$	$0.\ b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}$ with $b_{-4} = 1$
Truncated number	$0.\ b_{-1}\ b_{-2}\ b_{-3}$	$0.\ b-1\ b-2\ b_3 + 0.001$

Fig. 2.10.8 Rounding method of truncation

In this method, the error range is approximately $-1/2$ to $+1/2$ in the LS position of the retained bits and is symmetrical about 0 (Refer Fig. 2.10.9) and hence the result of rounding is a **unbiased approximation**.

Original Number	0.011011	0.011100
Truncated Number	0.011	0.100
	Negative error	**Positive error**

Fig. 2.10.9

As mentioned earlier, it is the most difficult method to implement because it requires an addition operation and a possible renormalization. In IEEE floating point standards by default rounding truncation is used.

Rounding can be implemented using only three guard bits. The first two of these bits are the two most significant bits of the section of the mantissa to be removed. The third bit is the logical OR of all bits beyond these first two bits in the full representation of the mantissa. This bit can be easily maintained during the intermediate steps of the operations to be performed. It should be initialized to 0. If a 1 is shifted out through this position, the bit becomes 1 and retains that value; hence it is usually called the **sticky bit**.

Review Questions

1. *State the rules for addition and subtraction.* **Dec.-06, Dec.-08, Marks 5**
2. *Write a note on problems in Floating point arithmetic.*
3. *Explain the working of floating point adder/subtractor.* **May-06, Marks 12**
4. *Explain the floating point Add/Subtract rules. With a detailed flowchart explain how floating point addition/subtraction is performed.* **Dec.-06, 12, Marks 13**
5. *Draw the hardware implementation of floating point operations.*
6. *How do you perform the floating point multiplication and division ?*
7. *What is the purpose of guard bits used in floating point operations ?* **May-06, Marks 2**
8. *What is Von Neumann rounding ?* **May-09, Marks 2**
9. *Describe how the floating point numbers are represented and used in digital arithmetic operations. Give an example.* **Dec.-07, Marks 16**
10. *Write rule for addition in floating point operation.* **Dec.-08, Marks 2**
11. *What are the problems in floating point arithmetic?* **May-10, Marks 16**

2.11 Arithmetic Logic Unit

Dec.-07, 08

The Arithmetic and Logic Unit (ALU) performs all the necessary adding, subtracting, shifting, and logical operations. It requires one or two operands upon which to operate and produces a result. An complexity of an ALU is determined by the way in which its arithmetic instructions are realized. Simple ALUs that perform fixed point addition and subtraction, as well as word based logic operations, can be realized by combinational circuits. ALUs that also perform multiplication and division can be constructed using circuit discussed in preceding sections for them. Let us see the simple ALU design with combinational circuits.

2.11.1 Design of Arithmetic Unit

The basic component of the arithmetic section of a simplest ALU is a parallel adder. A parallel adder can be constructed using number of full adder circuits connected in cascade. Let us see the implementation of various functions of arithmetic unit using parallel adder. Table 2.11.1 shows these implementations.

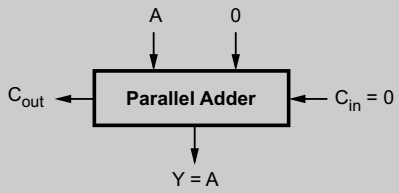
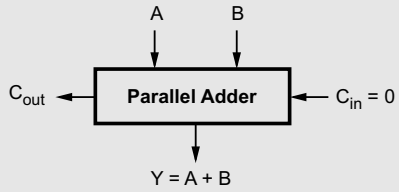
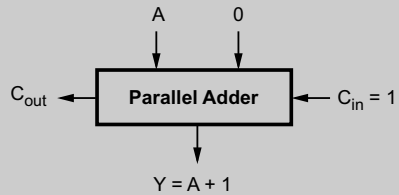
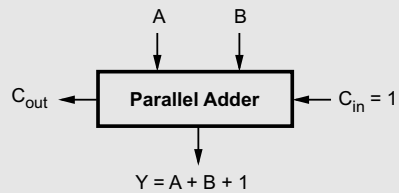
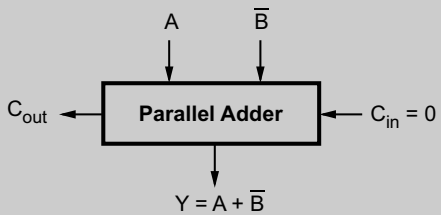
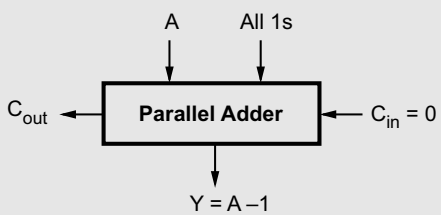
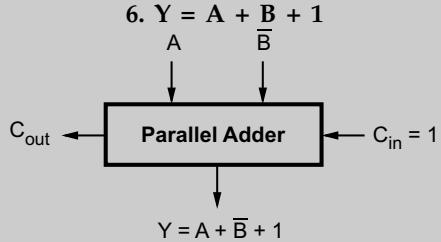
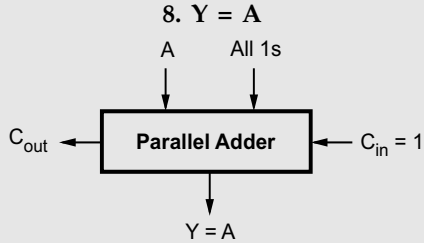
Function and Implementation	
1. Y ← A 	**3. Y ← A + B** 
2. Y ← A + 1 	**4. Y ← A + B + 1** 
5. Y ← A + $\overline{B}$ 	**7. Y = A – 1** 
6. Y = A + $\overline{B}$ + 1 	**8. Y = A** 

Table 2.11.1 Implementation of basic arithmetic operations

- Looking at Table 2.11.1 we can realize that by selecting proper B and C_{in} inputs we can implement basic arithmetic operations. There are four possible B inputs: true, complement, all 1s and all 0s and two possible C_{in} inputs 0 and 1.
- Referring Table 2.11.2, we can write truth table for B input with respect to S_0 and S_1 signals as follows :

S_1	S_0	B'
0	0	0
0	1	B
1	0	$\overline{B}$
1	1	1

Note - B′ indicates modified B input

Table 2.11.2 Truth table

Using the logic diagram of modified B input we can draw the logic diagram of arithmetic circuit, as shown in Fig. 2.11.2. (Fig. 2.11.2 see on next page)

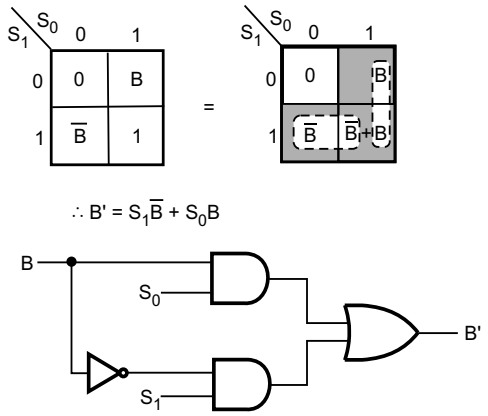


Fig. 2.11.1 K-map simplification and logic diagram of modified B input

Example 2.11.1 *Design an arithmetic circuit with one selection variable S and two n-bit data inputs A and B. The circuit generates the following four arithmetic operations in conjunction with the input carry C_{in}. Draw the logic diagram for the first two stages.*

S	$C_{in} = 0$	$C_{in} = 1$
0	D = A + B	D = A + 1
1	D = A – 1	D = A + $\overline{B}$ + 1

Solution :

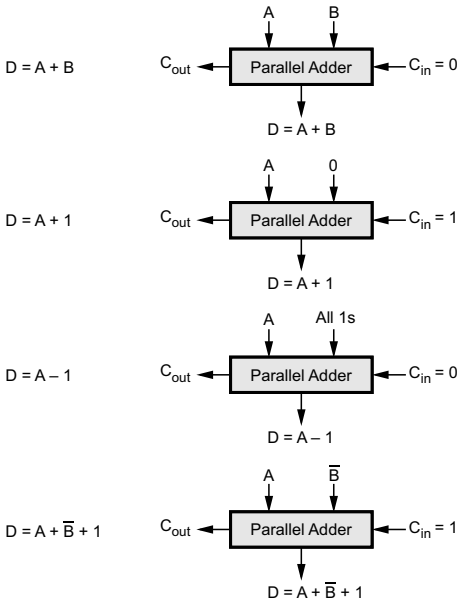


Fig 2.11.3

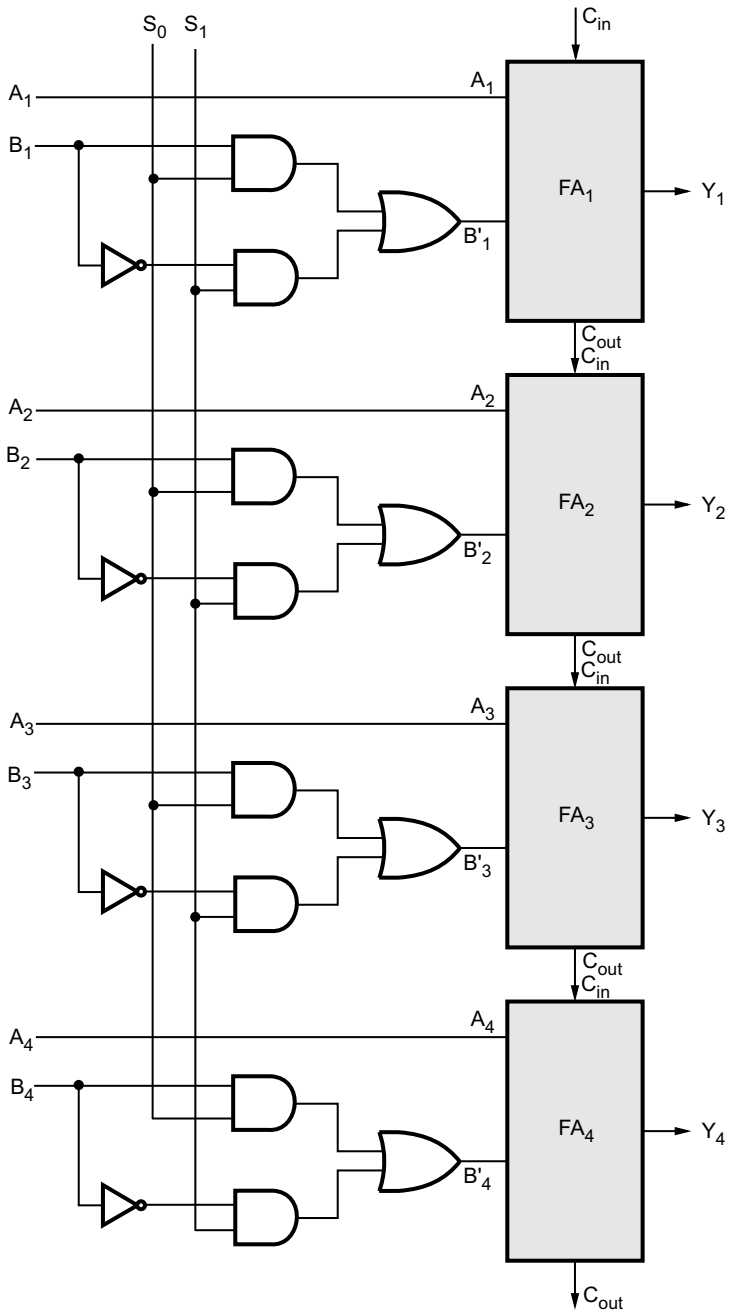


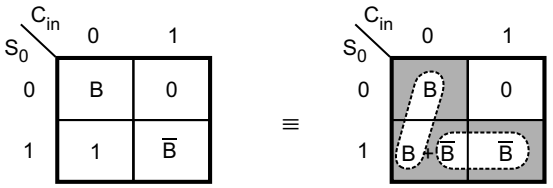
Fig. 2.11.2 Logic diagram of arithmetic circuit

Referring Fig. 2.11.3, we can write truth table for B input with respect to S_0 and C_{in} signals as shown in Table 2.11.3.

S_0	C_{in}	B input	Function
0	0	B	D = A + B
0	1	0	D = A + 1
1	0	1	D = A – 1
1	1	$\overline{B}$	D = A + $\overline{B}$ + 1

Table 2.11.3

K-map simplification



$\therefore B' = B\,\overline{C}_{in} + \overline{B}\,S_0$

Fig. 2.11.4

Using the logic diagram of modified B input we can draw the logic diagram of arithmetic circuit, as shown in the Fig. 2.11.7. (See on next page).

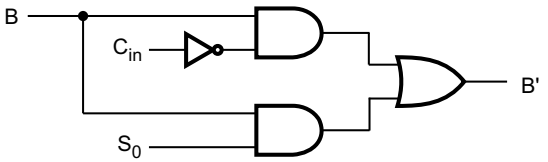


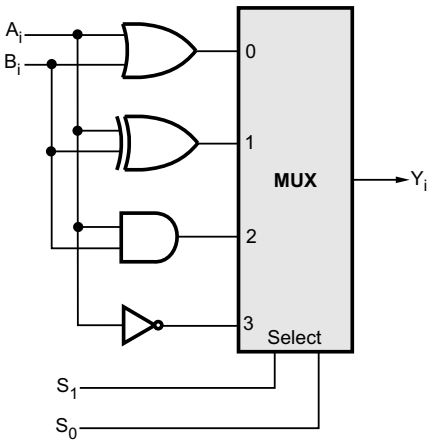
Fig. 2.11.5 Logic diagram of modified B input

2.11.2 Design of Logic Circuit

The design of logic circuit is comparatively simple than the design of arithmetic circuit. Fig. 2.11.6 shows the function table and the logic diagram for logic circuit.

S_1	S_0	Output	Function
0	0	Y = A ∨ B	OR
0	1	Y = A ⊕ B	XOR
1	0	Y = A ∧ B	AND
1	1	Y = $\overline{A}$	NOT

(a) Function table



(b) Logic diagram

Fig. 2.11.6 Logic circuit

Note : The logic diagram shown in Fig. 2.11.6 (b) is repeated n times for n-bit

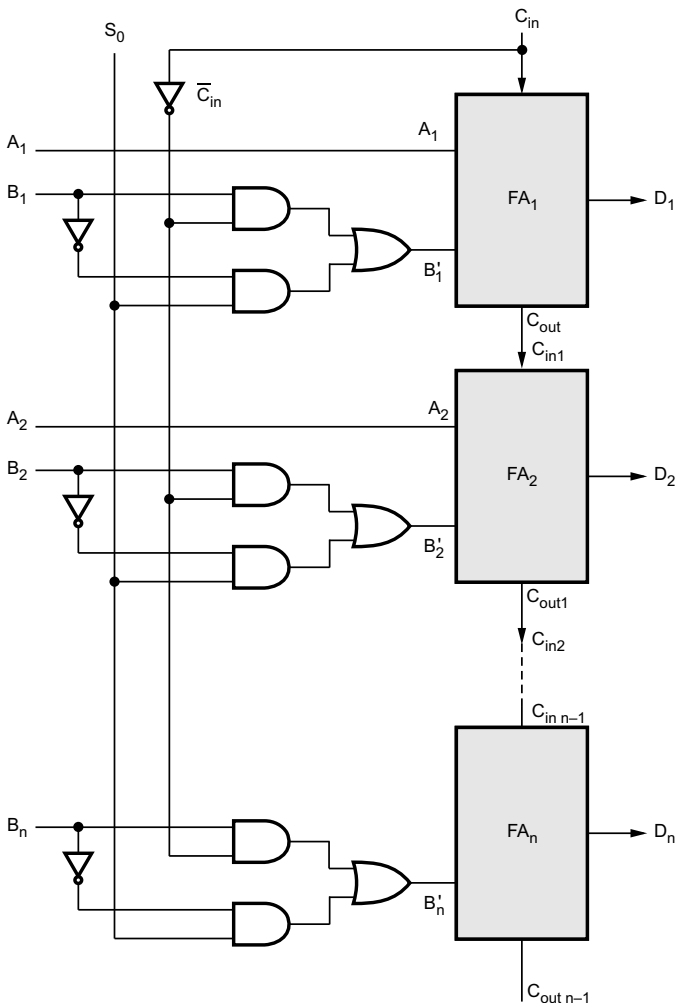


Fig. 2.11.7 Logic diagram of specified arithmetic circuit

logic circuit. Therefore, inputs A and B, and output Y are shown with subscript i, where i = 1, 2, … n.

We have seen the design of arithmetic and logic circuits for a simple ALU. By combining both circuits with the help of multiplexer we can get the arithmetic and logic circuit, as shown in the Fig. 2.11.8. When S (mode select) is 0, the output of

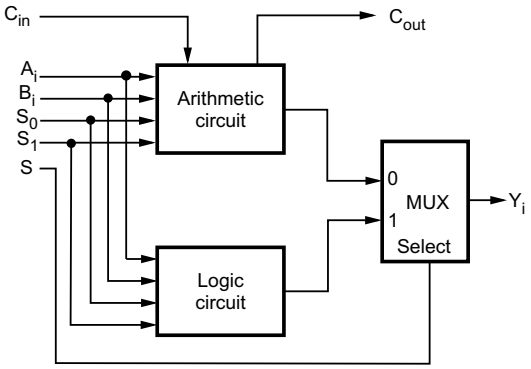


Fig. 2.11.8 Combining arithmetic and logic circuits

arithmetic circuit is transferred as a final output; otherwise the output of logic circuit is transferred as a final output.

2.11.3 Combining Arithmetic and Logic Units

The design of ALU in above mentioned way is the simplest and straight forward. A more efficient ALU can be implemented if we investigate the possibility of generating logic operations in an already available arithmetic circuit.

We know that carry input is not required in the logic circuits. Hence, when logic operation is selected (S = 1), the carry input must be zero. This gives us the output sum in full-adder circuit as

$$Y_i = A_i \oplus B_i \oplus 0 \qquad \because C_{in} = 0$$
$$= A_i \oplus B_i$$

Using this relation we have to change the outputs as shown in Table 2.11.4.

S	S_1	S_0	A'_i	B'_i	C_i	$F_i = A_i \oplus B_i$	Operation	Required operation
1	0	0	A_i	0	0	$Y_i = A_i$	Transfer A	OR
1	0	1	A_i	B_i	0	$Y_i = A_i \oplus B_i$	XOR	XOR
1	1	0	A_i	$\overline{B}_i$	0	$Y_i = A_i \odot B_i$	X-NOR	AND
1	1	1	A_i	1	0	$Y_i = \overline{A_i}$	NOT	NOT

Table 2.11.4 Excitation table for ALU

Let us see, how we can change these outputs.

Case 1 : S_1 S_0 = 00 Required operation OR

Here, $Y_i = A_i$. To make $Y_i = A_i \vee B_i$, we must change the input A'_i to each full adder circuit from A_i to $A_i + B_i$.

Case 2 : S_1 S_0 = 01 Required operation XOR

Here, $Y_i = A_i \oplus B_i$. Hence change is not required.

Case 3 : S_1 S_0 = 10 Required operation AND

Here, $Y_i = A_i \odot B_i$ and we want $Y_i = A_i B_i$. Let us investigate the possibility of ORing each input A_i with some boolean function K_i. Therefore, now

$$Y_i = (A_i + K_i) \oplus \overline{B}_i = (A_i + K_i) B_i + \overline{(A_i + K_i)}\,\overline{B}_i$$
$$= A_i B_i + K_i B_i + (\overline{A}_i \cdot \overline{K}_i)\overline{B}_i = A_i B_i + K_i B_i + \overline{A}_i \overline{K}_i \overline{B}_i$$

To have $Y_i = A_i B_i$, terms 2 and 3 must be 0. i.e. $K_i B_i + \overline{A}_i \overline{K}_i \overline{B}_i = 0$. This result can be obtained when $K_i = \overline{B}_i$. This is illustrated as follows :

When $K_i = \overline{B}_i$

$$Y_i = A_i B_i + \overline{B}_i B_i + \overline{A}_i \overline{\overline{B}_i} \overline{B}_i = A_i B_i + \overline{B}_i B_i + \overline{A}_i B_i \overline{B}_i$$

$$= A_i B_i \qquad \because \overline{B}_i B_i = 0$$

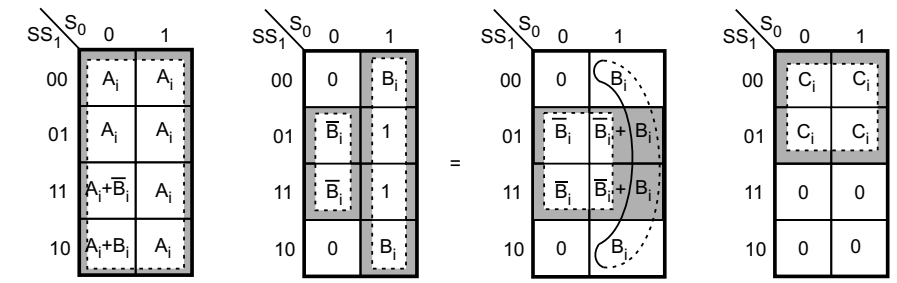
Case 4 : $S_1 S_0$ = 1 1 Required operation NOT

Here, $Y_i = \overline{A}_i$. Hence change is not required. With above changes we can write excitation table for ALU as shown in Table 2.11.5.

S	S_1	S_0	C_i'	A_i'	B_i'
0	0	0	C_i	A_i	0
0	0	1	C_i	A_i	B_i
0	1	0	C_i	A_i	$\overline{B}_i$
0	1	1	C_i	A_i	1
1	0	0	0	$A_i + B_i$	0
1	0	1	0	A_i	B_i
1	1	0	0	$A_i + \overline{B}_i$	$\overline{B}_i$
1	1	1	0	A_i	1

Table 2.11.5 Excitation table for ALU

K-map simplification for A_i', B_i' and C_i'



$A_i' = A_i + SS_1\overline{S}_0\overline{B}_i + S\overline{S}_1\overline{S}_0 B_i$ $B_i' = S_1 B_i + S_0 B_i$ $C_i' = \overline{S}C_i$

Fig. 2.11.9

Therefore, Boolean expressions for ALU inputs can be given as

$$A_i' = A_i + S S_1 \overline{S}_0 \overline{B}_i + S\overline{S}_1 \overline{S}_0 B_i$$

$$B_i' = S_1 \overline{B}_i + S_0 B_i$$

$$C_i' = \overline{S}C_i$$

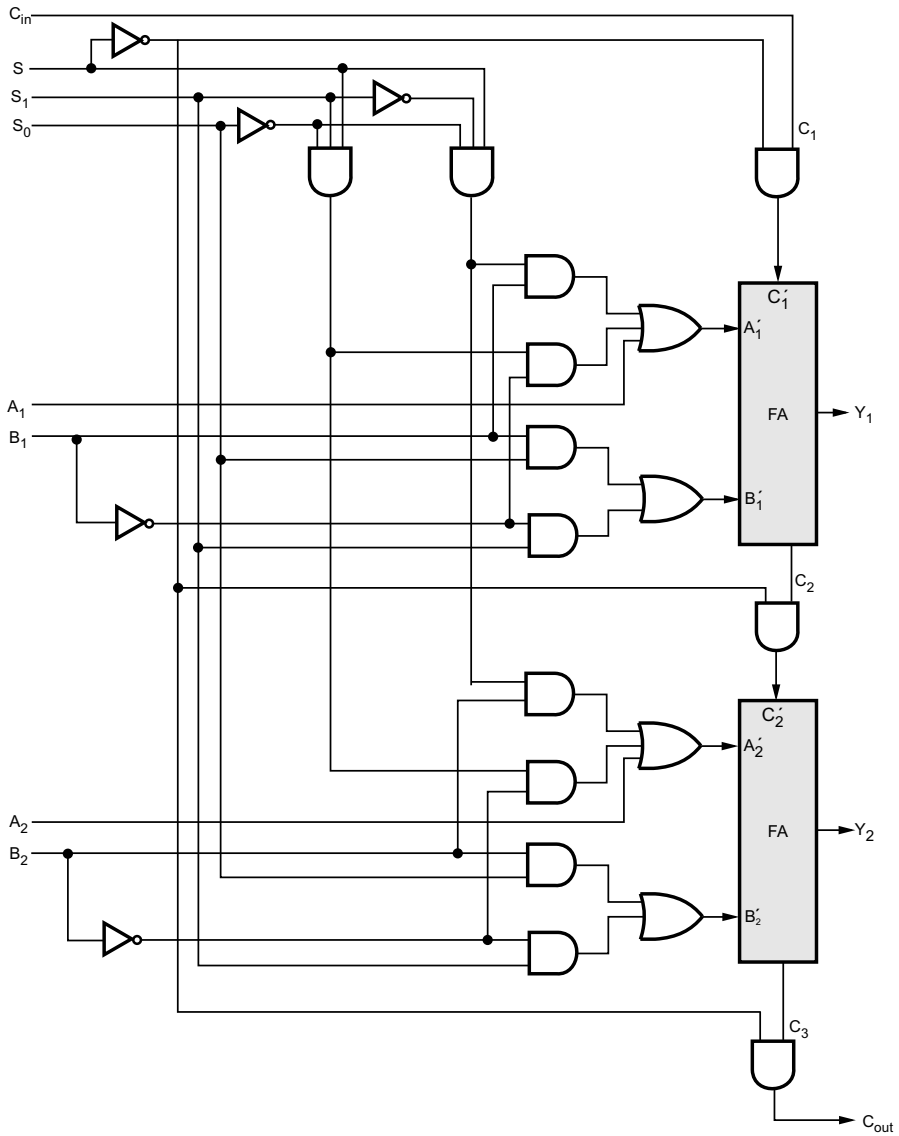


Fig. 2.11.10 Logic diagram of arithmetic logic unit (ALU)

When S = 1 , these equations reduce to

$$A_i' = A_i$$

$$B_i' = S_1 \overline{B}_i + S_0 B_i$$

$$C_i' = 0$$

which are the functions of arithmetic circuits. Fig. 2.11.10 shows the final ALU circuit. It shows only first two stages, but can easily be extended to more stages.

2.11.4 Status Register

The design of ALU also involves the design of status register. The status register indicates certain conditions of the result such as

1. Result is zero
2. Result is negative / positive
3. Result with carry
4. Result with overflow

These conditions are indicated by certain bit conditions. For example, if bit is 0, result is non zero and if bit is 1 result is zero. The bits which indicate conditions of the results are also called as flag bits. Let us see bits in the status register.

C-bit : Carry Bit. It is set when the output carry of ALU is 1; otherwise it is 0.

S-bit : Sign Bit. It is set when MSB (Most Significant Bit) of the result is 1; otherwise it is zero. It indicates the sign of the result. When S bit is 1, result is negative ; otherwise result is positive.

Z-bit : Zero Bit. It is set when the result is zero; otherwise it is zero.

V-bit : Overflow Bit. In 2's complement arithmetic, most significant bit is used to represent sign and remaining bits are used to represent magnitude of a number. It is set if the result of a signed operation is too large to fit in the number of bits available (7-bits for 8-bit number) to represent it. The overflow can be detected by exclusively - ORing two most significant carry out signals.

Fig. 2.11.11 shows the logic circuit which detects above mentioned conditions and sets the flag bits accordingly.

Using these status bits we can relate inputs A and B as follows. Let us consider A and B are unsigned 4-bit binary numbers then after operation A – B

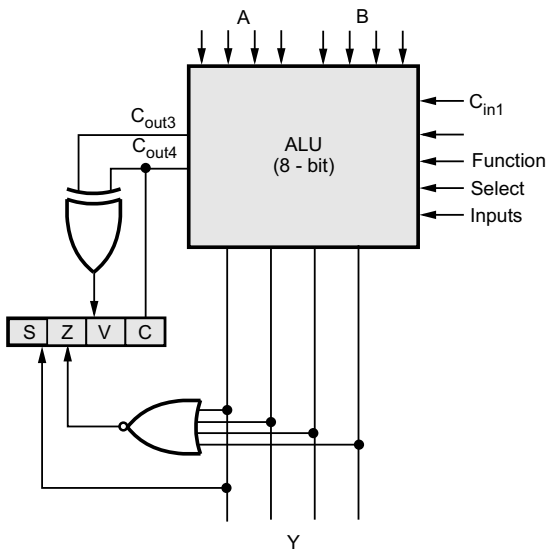


Fig. 2.11.11 Setting bits in a status register

Case 1 : When Z = 1, A = B

This is illustrated using this example.

Let us consider A = 0100 and B = 0100, i.e. A = B

```
∴          1          ← Carry
           0 1 0 0
      +    1 1 0 0    2's complement of 0 1 0 0
     ---------------
        1  0 0 0 0
```

F = 0 0 0 0, ∴ Z = 1

Case 2 : When Z = 0 and C = 1, A > B

This is illustrated using following example.

Let us consider A = 0 1 0 0 and B = 0 0 1 0, i.e. A > B

```
∴          1          ← Carry
           0 1 0 0
      +    1 1 1 0    2's complement of 0 0 1 0
     ---------------
        1  0 0 1 0
```

F = 0 0 1 0 ∴ Z = 0 and C = 1

Case 3 : When Z = 0 and C = 0, A < B

This is illustrated using following example.

Let us consider A = 0 0 1 0 B = 0 1 0 0, i.e. A < B

```
∴          0 0 1 0
      +    1 1 0 0    2's complement of 0 1 0 0
     ---------------
           1 1 1 0
```

F = 1 1 1 0 (2's complement form) ∴ Z = 0 and C = 0

Now consider A and B are signed 4-bit binary numbers.

Case 1 : When Z = 1, A = B

This is illustrated using following example. Let us consider A = 01 0 0 and B = 0 1 0 0, i.e. A = B.

```
∴          1          ← Carry
           0 1 0 0
      +    1 1 0 0    2's complement of 0 1 0 0
     ---------------
        1  0 0 0 0
```

F = 0 0 0 0 ∴ Z = 1 and S = 0

Case 2 : When Z = 0 and (S = 0,V = 0 or S = 1, V = 1), A > B

This is illustrated using following examples :

Example 1 :

A = 0 1 0 0 and B = 0 0 1 0. i.e. A > B

$\therefore$ 1 ← Carry

0 1 0 0

\+ 1 1 1 0 2's complement of 0 0 1 0

1 0 0 1 0

F = 0 0 1 0 $\therefore$ Z = 0, S = 0 and V = 1 $\oplus$ 1 = 0

Example 2 :

A = 0 1 0 0 and B = 1 0 1 1(– 5), i.e. A > B

$\therefore$ 1 ← Carry

0 1 0 0

\+ 0 1 0 1 2's complement of (– 5) 1 0 1 1

1 0 0 1

F = 1 0 0 1 $\therefore$ Z = 0, S = 1 and V = 0 $\oplus$ 1 = 1

Case 3 : S = 1, V = 0 or S = 0 ,V = 1, A < B

This is illustrated using following examples :

Example 1 :

A = 0 0 1 0 and B = 0 1 0 0, i.e. A < B

$\therefore$ 0 0 1 0

\+ 1 1 0 0 2's complement of 0 1 0 0

1 1 1 0

F = 1 1 1 0 Result is negative without overflow hence A < B

Example 2 :

A = 1 0 1 1 (– 5) and B = 0 1 0 0, i.e. A < B

$\therefore$ 1 0 1 1

\+ 1 1 0 0 2's complement of 0 1 0 0

1 0 1 1 1

F = 0 0 01 $\therefore$ S = 0, V = 0 $\oplus$ 1 = 1. Here, result is positive with overflow hence A < B.

The status bits and the relation between A and B can be summarized as shown in Table 2.11.6.

Relation	Condition of status bits for	
	Unsigned numbers	Signed numbers
A = B	Z = 1	Z = 1
A > B	Z = 0 and C = 1	Z = 0 and (S = 0,V = 0 or S = 1,V = 1)
A < B	Z = 0 and C = 0	S = 1,V = 0 or S = 0, V = 1

Table 2.11.6 Status bit and relation between A and B

Example 2.11.2 *Draw the structure of 4-bit MSI ALU circuit block.* **Dec.-07, Marks 2**

Solution : The Fig. 2.11.12 shows the structure of 4-bit MSI ALU-74181. Here, the 2's-complement addition is implemented by the carry look-ahead method. The modules labelled M_1 and M_2 generate a pair of 4-bit signals IP and IG that serve as internal propagate and generate, respectively, in the arithmetic mode and as minterm sources in the logic mode.

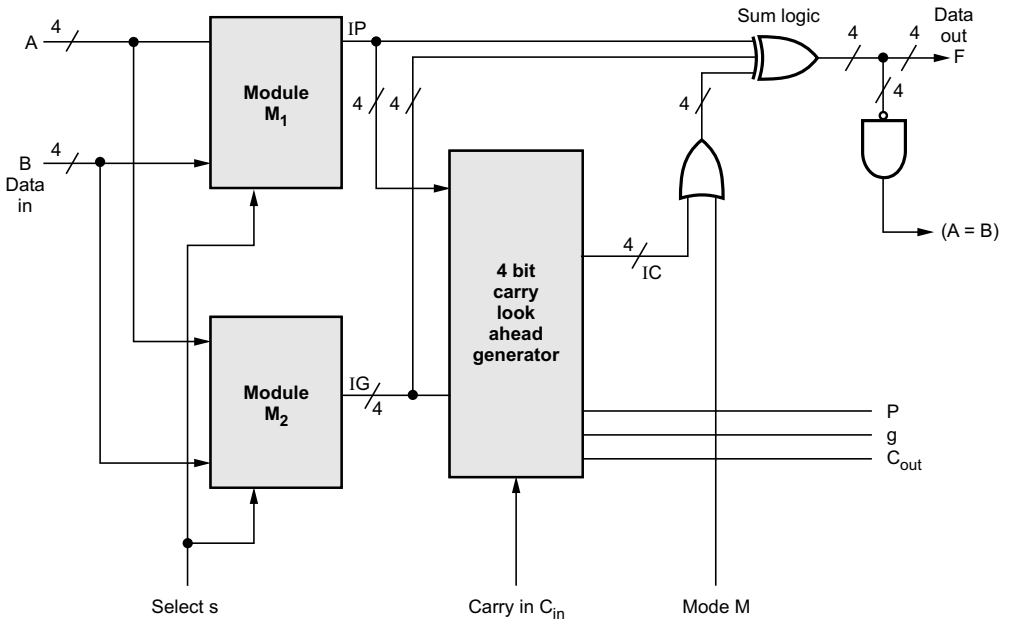


Fig. 2.11.12 Structure of 74181(MSI 4-bit ALU)

2.11.5 Sequential ALUs

Both multiplication and division can be implemented by combinational logic; however, it is generally impractical to merge these operations with addition and subtraction into a single, combinational ALU. Because combinational multipliers and dividers are costly in terms of hardware and they are also much slower than addition and subtraction circuits. An n-bit combinational multiplier or divider is typically

composed of n or more levels of add-subtract logic, making multiplication and division at least n times slower than addition or subtraction. The propagation delays of gates that involved in the multiply and divide logic also does the multiplication or division still slower. Thus, when n is not very small, it is preferrable to construct ALU with low cost sequential circuit where add and subtract each take one clock cycle, while multiplication and division are multicycle operations.

The Fig. 2.11.13 shows the basic structure of a sequential ALU. Here, three one-word registers are used for storage of operands : the accumulator (AC), the multiplier-quotient register (MQ), and the data register (DR). The AC and MQ can be organized to form two word register (register pair) AC · MQ. This register pair is capable of left and right shifting. The ALU is a combination of parallel adder and logic circuits. It is capable of performing addition, subtraction, and logical operations. The ALU derives its input from AC and DR and places its result in AC. The MQ register is called multiplier-quotient register because it stores multiplier during multiplication and quotient during division. DR stores the multiplicand or divisor, while the result is stored in the register pair AC · MQ. The case of multiplication AC · MQ stores product, and in case of division AC stores remainder and MQ stores quotient.

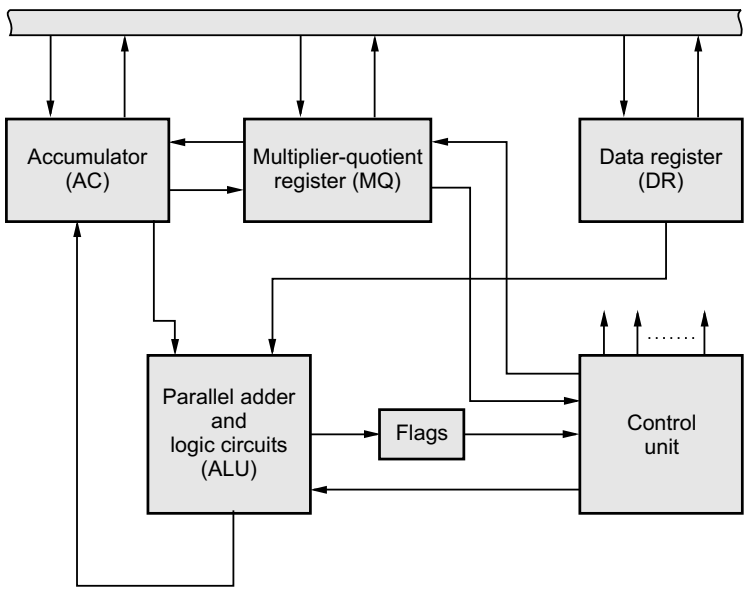


Fig. 2.11.13 Basic structure of sequential ALU

The Table 2.11.7 summarizes how the registers are used for different operations.

Operation	Registers used
Addition	AC ← AC + DR
Subtraction	AC ← AC – DR
Multiplication	AC · MQ ← DR × MQ
Division	AC · MQ ← MQ / DR
AND	AC ← AC AND DR
OR	AC ← AC OR DR
EX-OR	AC ← AC XOR DR
NOT	AC ← NOT (AC)

Table 2.11.7

2.11.6 ALU Expansion

It is possible to construct an entire sequential ALU for fixed number of bits, say m-bits on a single IC chip. Moreover, the ALU can easily be constructed to handle operands of size n = km or indeed any word size n > m in two ways :

- Spatial expansion
- Temporal expansion

Spatial Expansion : In this, K copies of the m-bit ALU are connected in the manner of a ripple carry adder to form a single ALU capable of processing km-bit words directly. The resulting ALU is commonly known as **bit-sliced ALU** because each component ALU concurrently processes a separate **slice** of m bits from each km-bit operand.

Temporal Expansion : In this, one copy of the m-bit ALU in the manner of serial adder performs an operation of km-bits words in K consecutive steps. This processing is commonly known as **multicycle** or **multi-precision processing**. In each step the m-bit ALU processes a separate m-bit slice of each operand.

We know that IC 74181 is a 4-bit ALU. The Fig. 2.11.14 shows how we can combine four such ALUs to construct a 16-bit bit-sliced combinational ALU.

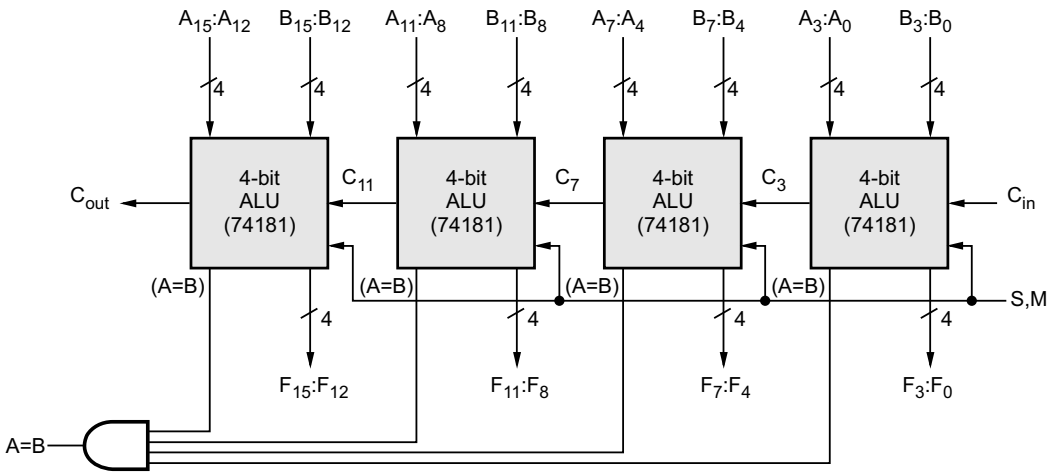


Fig. 2.11.14 16-bit bit-sliced combinational ALU

Review Questions

1. *State the function of ALU.*
2. *How different arithmetic and logic functions are realized and integrated in an ALU ? Explain.*

Dec.-08, Marks 16

3. *Explain the design of arithmetic unit.*
4. *Explain the design of logic circuit.*
5. *Explain the design of status register.*

2.12 Coprocessors

Dec.-10, 11

The instruction set of processor is not optimized to do complex numerical calculations, CRT graphics manipulation, or word processing. Complicated arithmetic operations like exponentiation and trigonometric functions are costly to implement in CPU hardware, while software implementations of these operations are slow. In order to implement these special functions with low-cost and fast hardware we have to use auxiliary processors called **arithmetic coprocessors** or simply **coprocessors**. In general, a coprocessor is a separate instruction-set processor. It has its own instruction set supporting the special exponential and trigonometric functions. The instructions and registers of coprocessor are just extension of the CPU's instruction set and registers. Both CPU and coprocessor execute their instructions from the same program. The instructions intended for the coprocessor are fetched by the CPU, and jointly decoded by both the CPU and the coprocessor, and executed by the coprocessor.

The Fig. 2.12.1 shows the typical coprocessor configuration. In coprocessor configuration both the CPU and coprocessor share entire memory and the I/O subsystem. They also share same bus control logic and clock generator. The coprocessor is connected to the CPU by several control lines that allow the activities of the two processors to be co-ordinated. In such a configuration, CPU is a master and coprocessor is a passive or slave device. The CPU can read or write in the coprocessor registers much similar manner as an external memory. When processor instructions are appeared in the program the communication between the CPU and coprocessor is initiated. After knowing the opcode of instruction to be executed and the operand's information, the coprocessor executes its instruction. It indicates its business by sending BUSY signal to the CPU. In mean time CPU is allowed to execute its instructions in the sequence whose execution is not dependent on the result of execution of instruction in process by coprocessor. When such instructions are executed by CPU and if BUSY signal is still active, it waits till the execution of instruction in coprocessor.

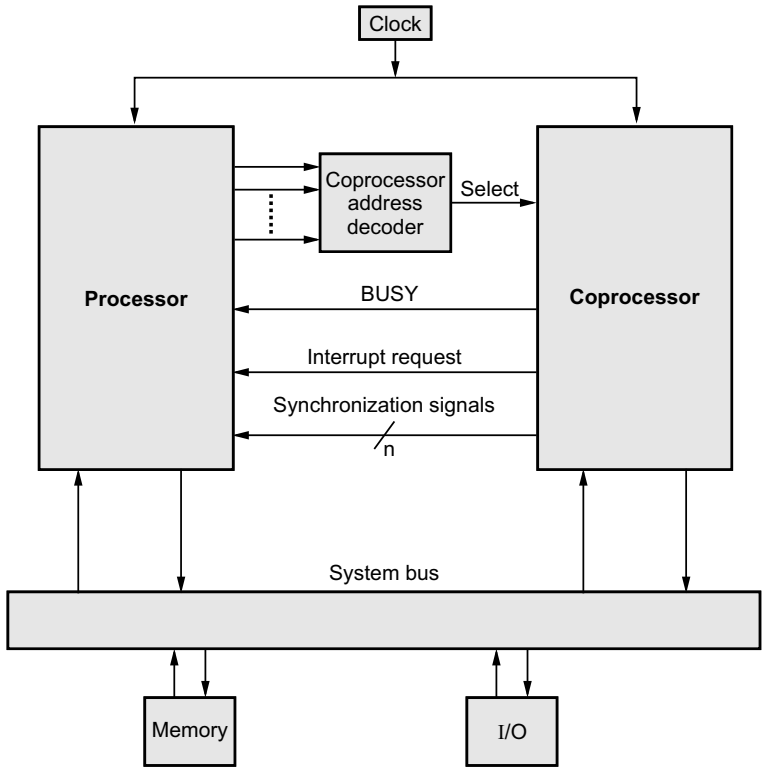


Fig. 2.12.1 Typical coprocessor configuration

When coprocessor is not connected in the system and coprocessor instructions are included in the program, the program control is transferred to a predetermined memory location where a software routine implementing the desired coprocessor instruction is

stored. This type of interrupt generated by CPU is called **coprocessor trap**. Thus the coprocessor approach makes it possible to provide either hardware or software support for certain instructions without altering the source or object code of the program being executed.

A coprocessor instruction typically consists of three fields : code, address and opcode. A code field distinguishes coprocessor instructions from other CPU instructions, the address field indicates the address of particular coprocessor when several coprocessors are used in the system, and finally the opcode field decides the operation to be executed by the coprocessor. The Table 2.12.1 shows the instruction set of the Motorola 68882 floating point coprocessor.

Type	Mnemonic	Description
Data transfer	FMOVE	Move word to/from coprocessor data or control register
	FMOVECR	Move word to/from ROM storing constants (0, 0 π, e, etc.)
	FMOVEM	Move multiple words to/from coprocessor
Data processing	FADD	Add
	FCMP	Compare
	FDIV	Divide
	FMOD	Modulo remainder
	FMUL	Multiply
	FREM	Remainder (IEEE format)
	FSCALE	Scale exponent
	FSGLMUL	Single-precision multiply
	FSGLDIV	Single-precision divide
	FSUB	Subtract
	FABS	Absolute value
	FACOS	Arc cosine
	FASIN	Arc sine
	FATAN	Arc tangent
	FATANH	Hyperbolic arc tangent
	FCOS	Cosine
	FCOSH	Hyperbolic cosine
	FETOX	e to the power of x
	FETOXMI	(e to the power of x) minus 1

	FGETEXP	Extract exponent
	FGETMAN	Extract mantissa
	FINT	Extract integer part
	FINTRZ	Extract integer part rounded to zero
	FLOGN	Logarithm of x to the base e
	FLOGNP1	Logarithm of x + 1 to the base e
	FLOG10	Logarithm to the base 10
	FLOG2	Logarithm to the base 2
	FNEG	Negate
	FSIN	Sine
	FSINCOS	Simultaneous sine and cosine
	FSINH	Hyperbolic sine
	FSQRT	Square root
	FTAN	Tangent
	FTANH	Hyperbolic tangent
	FTENTOX	10 to the power of x
	FTWOTOX	2 to the power of x
Program control	FBcc	Branch if condition code (status) cc is 1
	FDBcc	Test, decrement count, and branch on cc
	FNOP	No operation
	FRESTORE	Restore coprocessor state
	FSAVE	Save coprocessor state
	FScc	Set (cc = 1) or reset (cc = 0) a specified byte
	FTST	Set coprocessor condition codes to specified values
	FTRAPcc	Conditional trap

Table 2.12.1

The 68882 coprocessor is used with the 68020 CPU. While decoding the instruction, the 68020 CPU identifies the coprocessor instructions by their distinctive opcodes. After identifying a coprocessor instruction, the 68020 CPU "wakes up" the 68882 by sending it certain control signals. In response 68020 transmits the opcode to a predetermined location in the 68882 that serves as an instruction register. After receiving the opcode the instruction register 68882 decodes the instruction and begins its execution, which can proceed in parallel with other instructions executed within the CPU. When the

coprocessor needs to load or store operands required for instruction execution, it asks the CPU to carry out the necessary address calculations and data transfers.

The 68882 supports the IEEE 754 floating point number formats with certain multiple-precision extensions. It also supports a decimal floating point format. It consists of eight 80-bit, floating point data registers FP_0 : FP_7 and several 32-bit control registers, including instruction and status registers. The status or condition codes CC generated by the 68882 when executing floating-point instructions include invalid operation, overflow, underflow, division by zero, and inexact result. This coprocessor status is stored in the control register, which can be read by the host CPU at the end of an instruction execution to check the proper execution of the instruction.

Review Questions

1. *Write a short note on coprocessors.*
2. *What is a coprocessor ?* **Dec.-10, 11, Marks 2**

2.13 Pipeline Processing

Pipelining is a general technique for increasing processor throughput. It can be applied to the processor in two ways : Hardware pipelining and instruction pipelining. This section describes the hardware pipelining and instruction pipelining. The prime motto of hardware pipelining is to increase processor throughput without requiring large amounts of extra hardware. The hardware pipelining is applied to the design of the complex datapath units such as multipliers and floating-point adders.

Usually pipeline processor consists of a sequence of m data-processing circuits, called elements, stages or segments. These stages collectively perform a single operation on a stream of data operands passing through them. The processing is done part by part in each stage, but the final result is obtained only after an operand set has passed through the entire pipeline. The Fig. 2.13.1 shows the structure of pipeline processor. Each stage consists of two major blocks : Multiword input register and datapath circuit. The multiword input registers R_i, hold partially processed results as they move through the pipeline and they also serves as buffers that prevent neighbouring stages from interfering with one another. In each clock period the individual stages process its data and transfers its results to the next stage.

The m-stage pipeline processor shown in Fig. 2.13.1 can simultaneously process up to m independent sets of data operands. Thus when the pipeline is full, m separate operations are being executed concurrently, each in a different stage. This gives a new final result from the pipeline every clock cycle. If time required to perform single

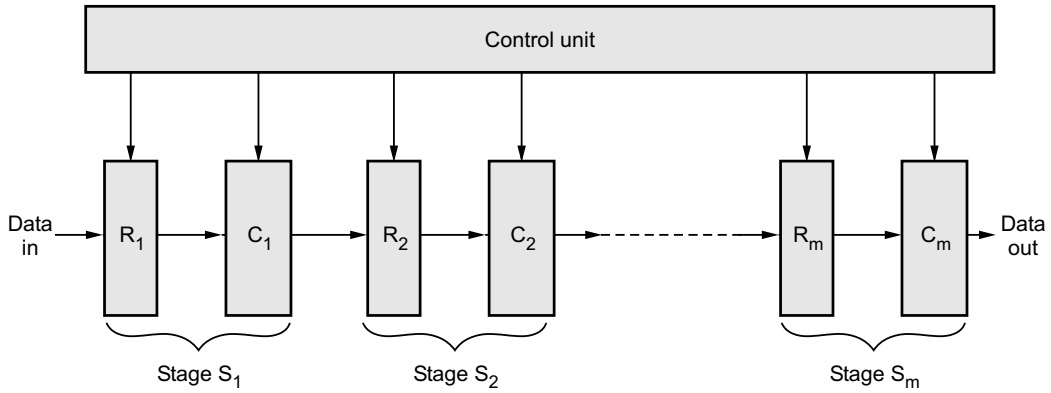


Fig. 2.13.1 Structure of pipeline processor

suboperation in the pipeline is T seconds then for m stage pipeline the time required to complete a single operation is mT seconds. This is called **delay** or **latency of the pipeline**. The maximum number of operations completed per second can be given as 1/T. This is called **throughput of the pipeline**.

The Fig. 2.13.2 (see Fig. 2.13.2 on next page) shows the four-stage floating point adder pipeline. In which, floating point addition operation is performed in four subsequent stages : Stage 1 - Exponent comparison, Stage 2 - Mantissa alignment, Stage 3 - Mantissa addition, Stage 4 - Normalization.

The Fig. 2.13.3 (see Fig. 2.13.3 on page 2-95) illustrates the behaviour of the adder pipeline when performing a sequence of N floating point additions. Here, N = 5 is the number of floating-point additions.

As shown in the Fig. 2.13.3, the S_1 gets first operand at t = 1, S_2 gets first operand at t = 2, S_3 gets first operand at t = 3 and so on. Due to this in m-stage pipelining for N operations we require N + (m – 1) time cycles. In our case, m = 4 and N = 5, therefore, 8 (5 + 3) time cycles will be required to perform 5 floating point addition operations. In general, we can specify N-stage pipeline speed-up as

$$S(m) = \frac{m \times N}{N + (m - 1)}$$

For 4-stage pipeline, it is given as

$$S(4) = \frac{4N}{N + 3}$$

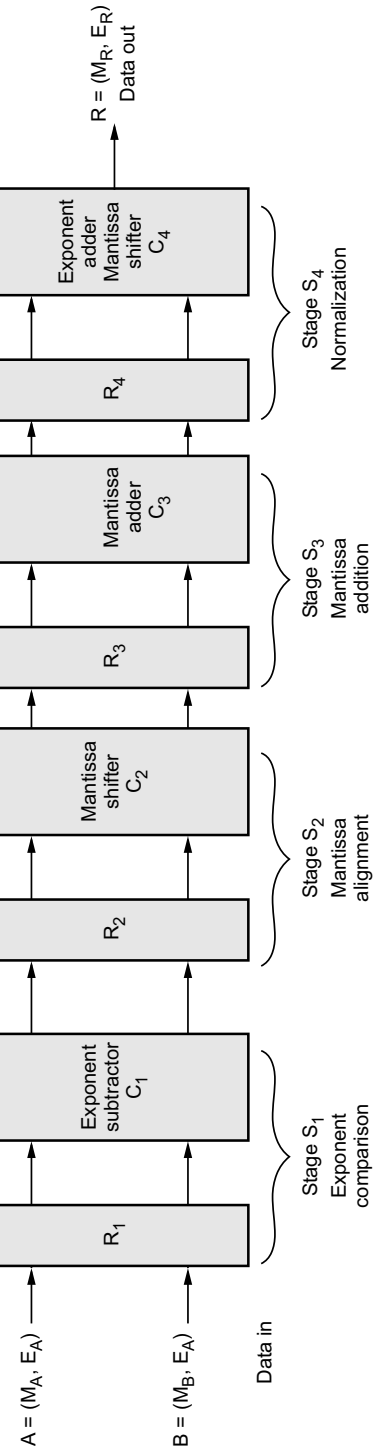


Fig. 2.13.2 Four stage floating-point adder pipeline

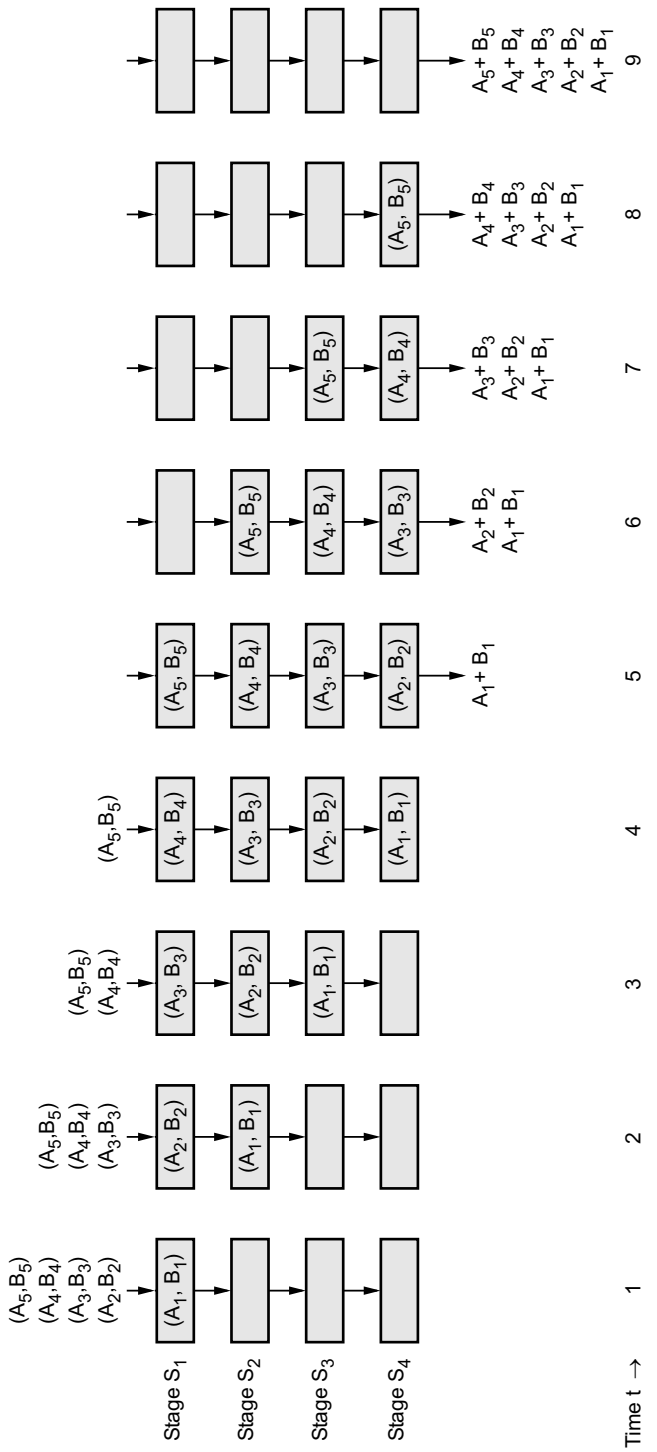

Stage S_1
Stage S_2
Stage S_3
Stage S_4
Time t →
1
2
3
4
5
6
7
8
9
(A_5,B_5) (A_4,B_4) (A_3,B_3) (A_2,B_2)
(A_1, B_1)
(A_2, B_2)
(A_3, B_3)
(A_4, B_4)
(A_5, B_5)
A_1+B_1
A_2+B_2
A_3+B_3
A_4+B_4
A_5+B_5

Fig. 2.13.3

Review Questions

1. *Give the structure of pipeline processor.*
2. *Define latency of the pipeline.*

2.14 Pipeline Design

Dec.-03, 11, May-06

In pipeline design we must consider the following points :

- To apply the pipeline we have to subdivide operation in multi-stage sequential suboperations.
- Suboperations should have approximately same execution time.
- Fast buffers must be placed between the stages to transfer data between the stages without interfering with one another and reliable transfer of data between stages.

The Fig. 2.14.1 shows the register level design of a floating-point adder pipeline based on the nonpipelined design of Fig. 2.10.2 and the four stage structure of Fig. 2.13.2. (See Fig. 2.14.1 on next page).

As shown in the Fig. 2.14.1, stage one does subtraction of two exponents to perform exponent comparison. Stage two does the alignment of mantissa with the help of result of comparison and shifter. Stage three does the addition or subtraction of mantissa depending on the add or subtraction operation. Stage four checks the leading zeroes in the mantissa result and adjust them to perform normalization. It also does the exponent adjustment required for normalization with the help of subtractor. Fix-point addition or subtraction can be performed using same circuit by bypassing stages 1, 2 and 4. Thus the circuit shown in Fig. 2.14.1 is an example of multifunction pipeline that can be configured either as a four-stage floating-point adder/subtractor or as a one-stage fixed-point adder.

One of the design aspect is to check whether or not a particular function or set of functions F should be implemented by a pipelined or nonpipelined processor. This can be analyzed as follows.

Let us assume that function F is divided into m independent sequential steps F_1, F_2, …, F_m so that it has an m-stage pipelined implementation P_m. Also let F_i be realizable by a logic circuit C_i with propagation delay (execution time) T_i and T_R be the delay of each stage S_i due to its buffer register R_i and associated control logic. We have seen that the computation time in each stage should be same. But many times it is practically not possible. In such cases the longest T_i times create bottleneck in the pipeline and force the faster stages to wait until the slower stages become available. Therefore, the minimum clock period of the pipeline, T_C is defined as

$$T_C = \max \{T_i\} + T_R \text{ for } i = 1, 2, \ldots, m$$

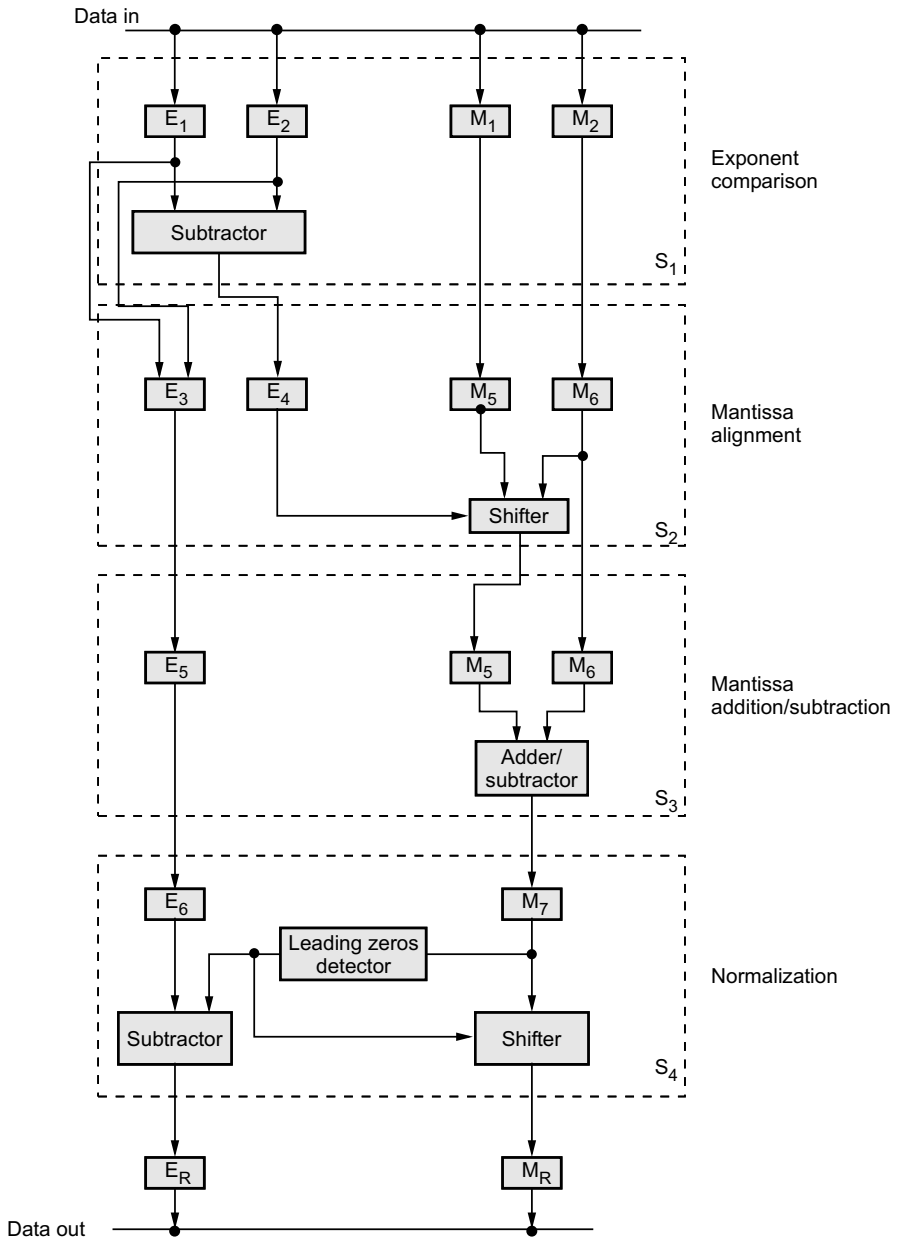


Fig. 2.14.1 Pipelined floating-point adder/subtractor

Thus,

The throughput of P_m is

$$\frac{1}{T_C} = \frac{1}{\max\{T_i\} + T_R}$$

On the other hand, a nonpipelined implementation P_1 of F has a delay of $\sum_{i=1}^{m} T_i$ or, equivalently, a throughput of $1/\left(\sum_{i=1}^{m} T_i\right)$. From above two equations we can conclude that the m-stage pipeline P_m has greater throughput than P_i, if

$$T_C < \sum_{i=1}^{m} T_i$$

Arithmetic Pipeline Design

Let us consider a floating point adder. We know that to perform floating point addition we have to do

- Exponent comparison and equalization
- Fraction addition
- Fraction normalization and exponent readjustment

These three task are divided into three different stages. The first stage for exponent comparison and equalization is implemented with an integer adder and some shift logic. The second stage for fraction addition is implemented using a high speed carry lookahead adder. The third stage for fraction normalization and exponent readjustment is implemented using a shifter and additional logic.

Shift registers are used to implement arithmetic or logical shift operations. To implement high speed adder, either a **carry propagation adder** (CPA) or **carry-save adder** (CSA) can be used.

In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry lookahead technique.

A CSA is used to add three input number and produce one sum output. In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector. Fig. 2.14.2 shows the addition of three binary numbers using CSA. As shown in the Fig. 2.14.2, in the first stage the CSA performs bitwise additions simultaneously on all columns of digits to produce two output numbers denoted as $s^b = (0, s_{n-1}, s_{n-2}, \ldots\ldots, s_1, s_0)$ and $c = (c_n, c_{n-1}, \ldots\ldots, c_1, 0)$.

Note that the leading bit in the bitwise sum S^b is always 0, and the tail bit of the carry vector C is always 0. The input-output relationships for S^b and C can be expressed as

$$S_i = x_i \oplus y_i \oplus z_i$$

$$C_{i+1} = x_i y_i \vee y_i z_i \vee z_i x_i$$

for i = 0,1, 2, ... n – 1, where $\oplus$ is the exclusive-OR and $\vee$ is the logical OR operation.

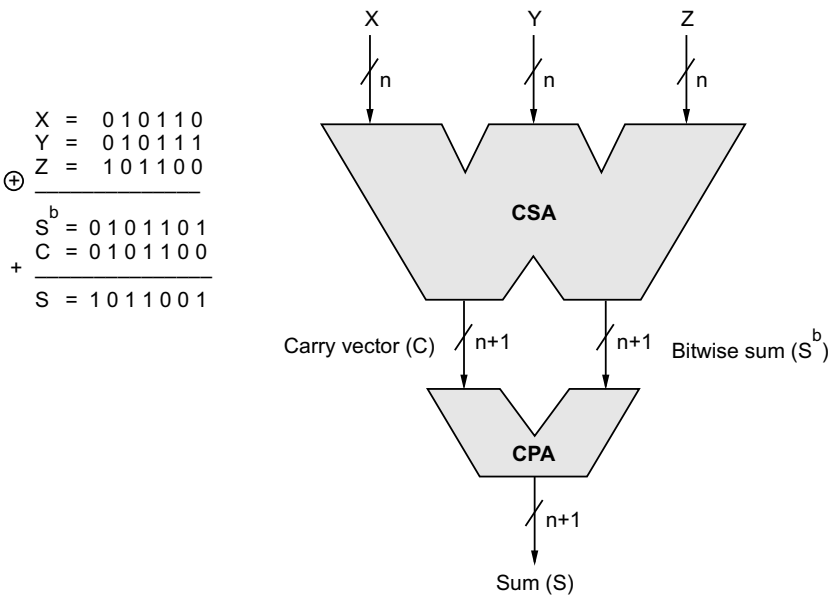


Fig. 2.14.2 Fast addition of three n-bit numbers using CSA and CPA

In the second stage the two output numbers, i.e. S^b and C are added using a CPA to get final sum, S.

Multiply Pipeline Design

Let us consider the multiplication of two 8-bit integer numbers A × B = P, where P is a 16-bit product. This multiplication can be represented as summation of eight partial products as shown below :

$P = A \times B = P_0 + P_1 + P_2 + P_3 + \ldots\ldots\ldots\ldots + P_7$.

	1 0 1 0 1 1 0 1	=	A
×	1 0 0 1 0 1 0 1	=	B
	1 0 1 0 1 0 1	=	P_0
	0 0 0 0 0 0 0 0	=	P_1
	1 0 1 0 1 1 0 1 0 0	=	P_2
	0 0 0 0 0 0 0 0 0 0 0	=	P_3
	1 0 1 0 1 1 0 1 0 0 0 0	=	P_4
	0 0 0 0 0 0 0 0 0 0 0 0 0	=	P_5
	0 0 0 0 0 0 0 0 0 0 0 0 0 0	=	P_6
	1 0 1 0 1 1 0 1 0 0 0 0 1 0 0 0	=	P_7
	0 1 1 0 0 1 0 0 1 0 1 1 0 0 0 1	=	P

The partial product P_i is obtained by multiplying the multiplicand A by the ith bit of B and then shifting the result i bits to the left for i = 0,1,2, ..., 7. The partial products and summation of eight partial products are carried out in different stages as given below :

Stage 1 : Generates all partial products, ranging from 8-bits to 15 bits

Stage 2 : In second stage four CSAs are used to merge eight numbers into four numbers ranging from 13 to 15 bits.

Stage 3 : The third stage consists of two CSAs, which merges four numbers into two 16-bit numbers.

Stage 4 : Two 16-bit numbers are added using CPA to get final product P.

These stages are illustrated in Fig. 2.14.3 and it is called **wallace tree of CSAs** plus a CPA.

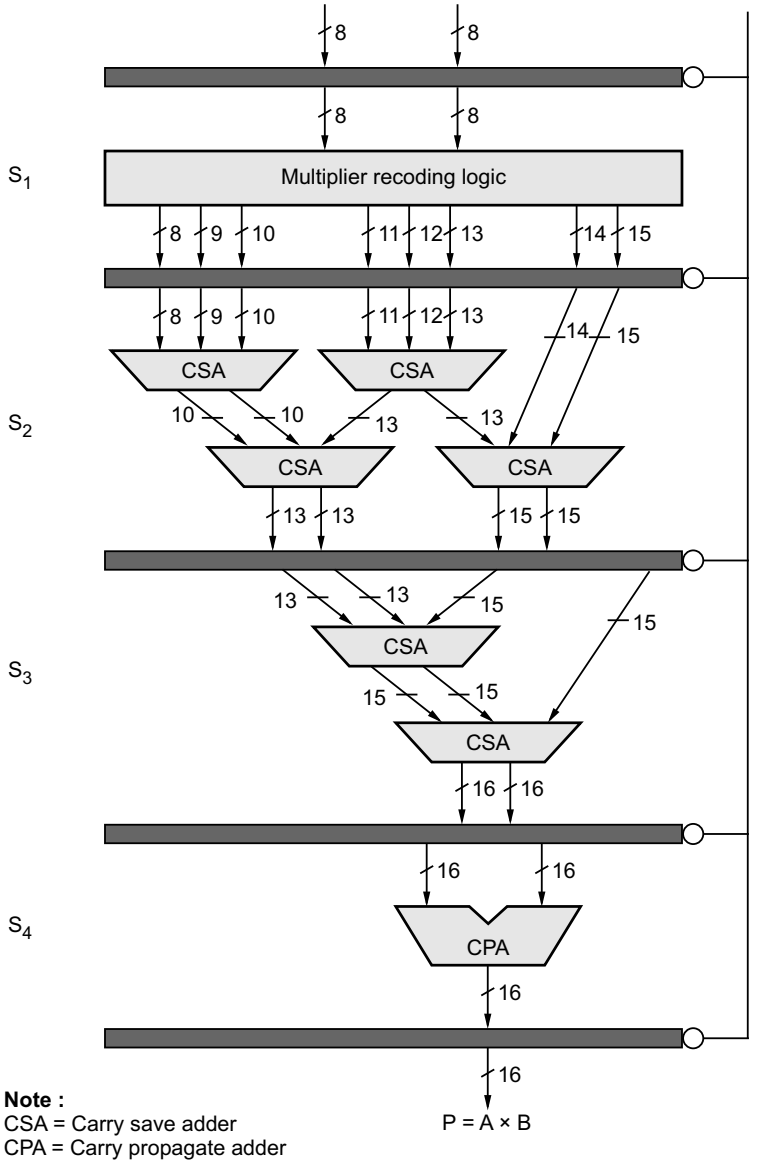


Fig. 2.14.3

Review Questions

1. *Draw and explain the four-stage floating point adder pipeline.*
2. *Explain the design aspects in implementation of pipelining.*
3. *Draw and explain the register level implementation of floating-point adder/subtractor.*
4. *Describe the basic structure of a pipeline processor and explain how it is carried out in a floating point adder.* **Dec.-03, Marks 16**
5. *What is the purpose of guard bits used in floating point operations ?* **May-06, Marks 2**
6. *Explain floating point adder pipeline with neat block diagram.* **Dec.-11, Marks 16**

Two Marks Questions with Answers

Q.1 ***Define half adder and full adder.***

Ans. :

- **Half adder :** The logic circuit which performs the arithmetic sum of two bits is called a half adder.
- **Full adder :** The logic circuit which performs the arithmetic sum of 3 bits (bit 1 : input 1, bit 2 : input 2, bit 3 : carry from the previous addition) is called a full adder.

Q.2 ***Define half subtractor and full subtractor.***

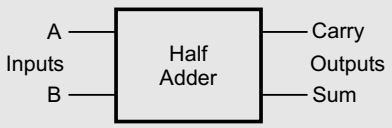
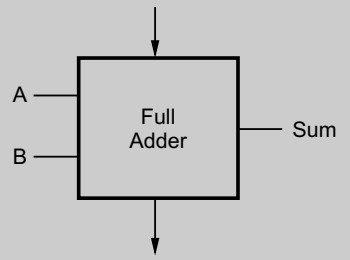
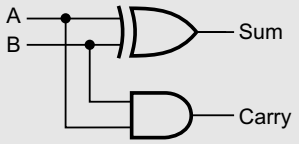
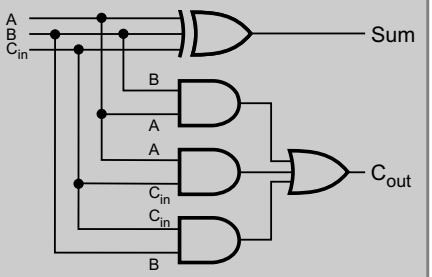
Ans. :

- **Half subtractor :** It is a combinational circuit that subtracts two bits and produces their difference and borrow.
- **Full subtractor :** It is a combinational circuit that performs a subtraction between 2 bits. It also takes into account borrow of the lower significant stage.

Q.3 ***What is the difference between half adder and full adder?*** **Dec.-07**

Ans. :

Sr. No.	Half adder	Full adder
1.	Half-adder takes two binary-inputs i.e. augend and addend bits and gives out two binary outputs as sum and carry.	Full-adder, alongwith augend and addend takes third additional bit Cin as input. Cin represents the carry from the previous lower significant position.
2.	Half-adder is not used in practice	Full adder is used in practice

3.	**Block diagram :** 	**Block diagram :** 
4.	**Logic diagram :** 	**Logic diagram :** 

Q.4 *What do you mean by carry propagation delay ?*

Ans. : **Definition :** In parallel adders, sum and carry outputs of any stage cannot be produced until the input carry occurs. This time delay in the addition process is called **carry propagation delay.** This delay increases with increase in the number of bits to be added in an adder circuit.

Q.5 *Draw the half adder circuit.* **May-09**

Ans. : Refer section 2.3.1.

Q.6 *Write the logic equations of a binary half a adder.* **May-11**

Ans. : Refer section 2.3.1.

Q.7 *Draw a full-adder circuit and give the truth table.* **May-09**

Ans. : Refer section 2.3.2.

Q.8 *Draw the full adder circuit using two half adders.* **Dec.-07**

Ans. : Refer section 2.3.2.

Q.9 *Draw the symbolic representation of the full-adder and give the expression for the sum.* **Dec.-06**

Ans. : Refer section 2.3.2.

Q.10 *What is a ripple carry adder ?* **May-07**

Ans. : The n-bit parallel adder using n number of full-adder circuits connected in cascade, i.e. the carry output of each adder is connected to the carry input of the next higher-order adder is called ripple carry adder.

Q.11 *Define overflow rule in addition.* **Dec.-09**

Ans. : When both operands a and b have the same sign, an overflow occurs when the sign of result does not agree with the signs of a and b.

Q.12 *What is a carry look-ahead adder ?*

Ans. : The method of speeding up the addition process by eliminating inter-stage carry delay is called lookahead-carry addition and the circuit that performs such addition is called carry look-ahead adder.

Q.13 *How do you ascertain an adder as "fast" ?* **Dec.-08**

Ans. : If adder has a mechanism such as lookahead carry adder to avoid carry propagation delay, the adder is called fast adder.

Q.14 *Discuss the principle behind the Booth's multiplier.* **May-08**

Ans. : Booth's algorithm generates a 2n-bit product and treats both positive and negative numbers uniformly. This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.

Q.15 *Discuss the role of Booth's algorithm in the design of fast multipliers.* **May-08**

Ans. : To speed-up the multiplication process in the Booth's algorithm a technique called **bit-pair recoding** is used. It is also called **modified Booth's algorithm**. It halves the maximum number of summands. In this technique, the Booth-recoded multiplier bits are grouped in pairs. Then each pair is represented by its equivalent single bit multiplier reducing total number of multiplier bits to half.

Q.16 *What is combinational multiplier or array multiplier ?*

Ans. : The multiplier which uses n shifts and adds operations to multiply n-bit binary number is called combinational multiplier or **array multiplier.**

Q.17 *Write rule for addition in floating point operation.* **Dec.-08**

Ans. : Refer section 2.10.1.

Q.18 *State the rules of floating point multiplication.*

Ans. :

1. Add the exponents and subtract bias. (127 in case of single precision numbers and 1023 in case of double precision numbers).

2. Multiply the mantissas and determine the sign of the result.

3. Normalize the result.

Q.19 ***State the rules of floating point division.***

Ans. : 1. Subtract exponents and add bias (127 in case of single precision numbers and 1023 in case of double precision numbers).

2. Divide the mantissas and determine the sign of the result.

3. Normalize the result.

Q.20 ***What do you mean by guard bits?***

Ans. : The mantissas of initial operands and final results are limited to 24-bits, including the implicit leading 1. But if we provide extra bits in the intermediate steps of calculations we can get maximum accuracy in the final result. These extra bits used in the intermediate calculations are called **guard bits**.

Q.21 ***State the commonly used methods of truncation.***

Ans. : There are three commonly used methods of truncation :

- Chopping
- Von Neumann rounding
- Rounding

Q.22 ***What are the two techniques for speeding up the multiplication operation?***

Ans. : i) Bit Pair recoding ii) Carry save addition

Q.23 ***How bit pair recoding of multiplier speeds up the multiplication process?***

Ans. : It guarantees that the maximum number of summands that must be added is n/2 for n-bit operands.

Q.24 ***How CSA speeds up multiplication?***

Ans. : It reduces the time needed to add the summands. Instead of letting the carries ripple along the rows, they can be saved and introduced into the next row, at the correct waited position.

Q.25 ***Write down the steps for restoring division.***

Ans. : 1. Shift A and Q left one binary position.

2. Subtract divisor from A and place answer back in A ($A \leftarrow A - B$).

3. If the sign bit of A is 1, set Q_0 to 0 and add divisor back to A (that is, restore A); Otherwise, set Q_0 to 1.

4. Repeat steps 1, 2, and 3 n times.

Q.26 *Write down the steps for non-restoring division*

Ans. :

Step 1 : If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A; otherwise, shift A and Q left and add divisor to A.

If the sign of A is 0, set Q_0 to 1; otherwise, set Q_0 to 0.

Step 2 : Repeat steps 1 and 2 for n times.

Step 3 : If the sign of A is 1, add divisor to A.

Q.27 *What is the advantage of non restoring over restoring division?*

Ans. : Non restoring division avoids the need for restoring the contents of register after an successful subtraction.

Q.28 *Draw the block diagram of an n-bit 2' s complement adder-subtractor.*

Dec.-03

Ans. : Refer section 2.3.5.

Q.29 *What is the purpose of guard bits used in floating point operations ?*

May-06

Ans. : Refer section 2.10.6.

Q.30 *Draw the symbolic representation of the full-adder and give the expression for the sum.*

Dec.-06

Ans. : Refer section 2.3.2.

Q.31 *Why floating point number is more difficult to represent and process than integer ?*

May-07

Ans. : In floating point numbers we have to represent any number in three fields sign, exponent and mantissa. The IEEE 754 standard gives the format for these fields and according to format the numbers are to be represented. In case of any process we have to consider mantissa and exponent separately. Therefore, floating point numbers are more difficult to represent and process than integer.

Q.32 *Discuss the principle behind the Booth's multiplier.*

May-07

Ans. : Refer section 2.7.

Q.33 *A memory byte location contains the pattern 00101100. What does this pattern represent when interpreted as a number? What does it represent as an ASCII Code ?*

Dec.-07

Ans. : 2CH, $(44)_{10}$. It represents ASCII code for comma (,)

Q.34 *What do you mean by end-around carry correction ?* **May-08**

Ans. : In binary subtraction using 1's complement method, the carry is generated after addition of two numbers (minuend + 1's complement of subtrahend) is called end-around carry. It is necessary to add this carry in the addition to get the correct result. Such a correction is called end-around carry correction.

Q.35 *Discuss the role of Booth's algorithm in the design of fast multipliers.* **May-08**

Ans. : Refer section 2.8.

Q.36 *What is Von Neumann rounding ?* **May-09**

Ans. : Refer section 2.10.6.

Q.37 *Draw the logical circuit for array multiplication.* **May-10**

Ans. : Refer section 2.8.2.

Q.38 *When does imprecise exception arise?* **May-10**

Ans. : An imprecise exception is usually a catastrophic failure, where the processor cannot continue processing or allow a particular program or task to continue. It may arise because of two possibilities :

1. The pipeline may have already completed instructions that are later in program order than the instruction causing the exception.
2. The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.

Q.39 *What is a coprocessor?* **Dec.-10**

Ans. : Refer section 2.12.

Q.40 *What is a datapath?* **Dec.-10**

Ans. : Refer section 2.1.

Q.41 *Write down the equations for carry generate and propagate.* **Dec.-11**

Ans. : $P_i = A_i \oplus B_i$

$G_i = A_i B_i$

Q.42 *Add (+ 7) with (– 3) in binary* **May-12**

Ans. :

0	1	1	1	**+ 7**
0	0	1	1	**+ 3**
1	1	0	0	**1's complement of 3**
1	1	0	1	**2's complement of 3 (– 3)**

	1	1	1	1	Carry
		0	1	1	1	+ 7
	+	1	1	0	1	(– 3)
Discard Carry	~~1~~	0	1	0	0	Result

$\therefore (+7)_{10} + (-3)_{10} = (0100)_2$

Q.43 *Subtract (– 5) from (–7) in binary* **May-12**

Ans. : Operation : (–7) – (– 5) = (– 7) + (5)

0	1	0	1	+ 5
0	1	1	1	+ 7
1	0	0	0	1's complement of 7
1	0	0	1	2's complement of 7

		1		Carry
1	0	0	1	(– 7)
0	1	0	1	+ 5
1	1	1	0	No carry, result is in 2's complement form

	0	0	0	1	1's complement of result
+				1	
	0	0	1	0	2's complement of result

$\therefore$ (– 7) + (5) = –2

Q.44 *What are the two approches to reduce the delay in address ?* **Dec.-12**

Ans. : The two approaches to reduce the delay in adders are :

- **Carry lookahead :** It reduces the delay of adders from increasing linearly with the word size to increase as the logarithm of the word size.
- **Carry select adder :** It divides the words to be added into blocks and forms two sums for each block in parallel (one with a carry in of zero and the other with a carry in of one). The carry output from the previous block controls a multiplexer that selects the appropriate sum.

Q.45 *What are the two techniques to speed up the multiplication operation ?* **Dec.-12, Marks 2**

Ans. : Refer section 2.8.

❑❑❑

3 Control Design

Syllabus

Hardwired control, Microprogrammed control, Multiplier control unit, CPU control unit, Pipeline control, Instruction pipelines, Pipeline performance, Superscalar processing, Nano programming.

Contents

3.1 *Introduction*

3.2 *Some Fundamental Concepts* ***Dec.-07, 08, 10***
. . . . . ***May-06, 07,08,*** · · · · · · · · · · *Marks 10*

3.3 *Hardwired Control* ***May-07, 10, 11,***
. . . . . ***Dec.-03, 06, 07, 08,*** · · · · · · · *Marks 16*

3.4 *Microprogrammed Control* ***May-06, 07, 08, 09, 10, 11, 12***
. . . . . ***Dec.-06, 07, 09, 12,*** · · · · · · · *Marks 16*

3.5 *Comparison between Hardwired and Microprogrammed Control*
. . . . . ***May-07, 12,*** · · · · · · · · · · · · · *Marks 2*

3.6 *Bit Slicing and Bit Sliced Microprogram Sequencer*
. . . . . ***May-11,*** · · · · · · · · · · · · · · · · · *Marks 16*

3.7 *Applications of Microprogramming*

3.8 *Pipeline Control* ***Dec.-03,06,07,08,10,11,12***
. . . . . ***May-06, 07, 08, 09, 10, 12*** · · *Marks 16*

3.9 *Superscaling Processing* . . . . . ***Dec.-11, 12,*** · · · · · · · · · · · · · *Marks 8*

3.10 *Nano Programming* . . . . . ***May-11, Dec.-11,*** · · · · · · · · · *Marks 16*

Two Marks Questions with Answers

3.1 Introduction

Generally, a processor unit is separated into two parts : Data processing unit and control unit. The data processing unit is a collection of functional units capable of performing certain operations on data, whereas control unit issues control signals to the data processing part to perform operations on data. These control signals select the functions to be performed at specific times and route the data through the appropriate functional units. In other words, the control unit decides the sequence in which certain sets of micro-operations to be performed to carry out particular operation on data. Hence, we can say that, the data processing unit is logically reconfigured by the control unit and it is intimately involved in the sequencing and timing of the data processing unit.

In this chapter, we are going to study the implementation of the control unit part of a processor using two basic design approaches : hardwired and microprogrammed. Before going to study these design approaches we will see functional requirements of processor unit and how various elements of CPU are controlled by control unit to provide these functions.

3.2 Some Fundamental Concepts

Dec.-07, 08, 10, May-06, 07, 08

Before going to discuss design approaches for processor unit we first get familiar with the functions of it. The primary function of a processor unit is to execute sequence of instructions stored in a memory, which is external to the processor unit. The sequence of operations involved in processing an instruction constitutes an instruction cycle, which can be subdivided into three major phases : Fetch cycle, decode cycle and execute cycle. This is illustrated in Fig. 3.2.1.

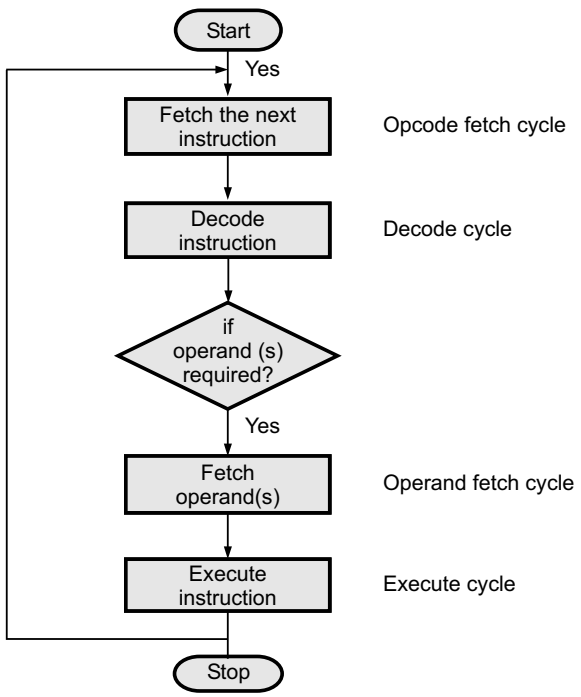


Fig. 3.2.1 Basic instruction cycle

To perform fetch, decode and execute cycles the processor unit has to perform set of operations called *micro-operations*. In this section, we are going to see how events of any instruction cycle can be described as a sequence of such micro-operations and

how the concept of micro-operations serves as a guide to the design of the processor unit.

Before we study these micro operations in detail, we examine the structure of the main data paths inside the processor unit. The Fig. 3.2.2 shows the single bus organisation of processor unit. It shows how the building blocks of processor unit are organised and how they are interconnected. They can be organised in a variety of ways. The Fig. 3.2.2 shows one organisation, in which the arithmetic and logic unit and all processor registers are connected through a single common bus. It also shows the external memory bus connected to memory address (MAR) and data register (MDR).

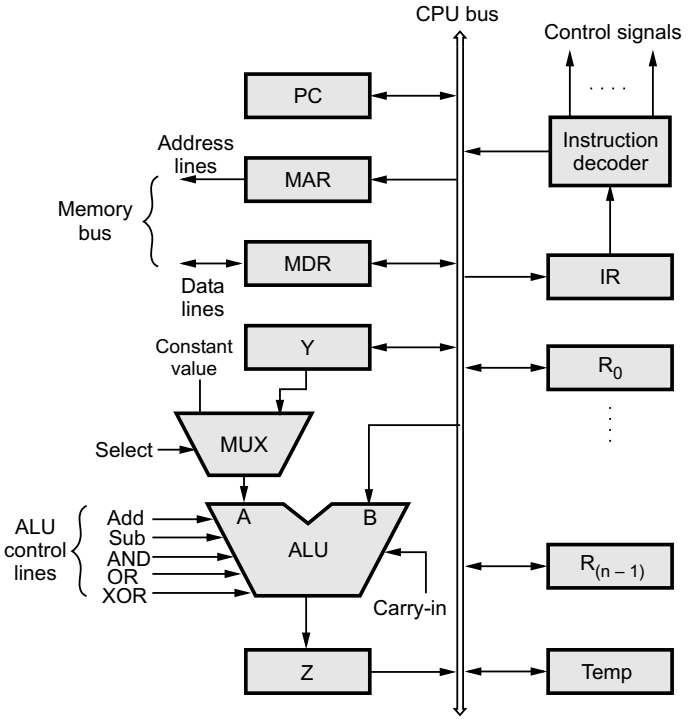


Fig. 3.2.2 Single bus organisation of processor

The registers Y, Z and Temp in Fig. 3.2.2, are used only by the CPU unit for temporary storage during the execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction. The programmer can not access these registers. The IR and the instruction decoder are integral parts of the control circuitry in the CPU unit. All other registers and the ALU are used for storing and manipulating data. The data registers, ALU and the interconnecting bus is referred to as **data path**.

Register R_0 through $R_{(n-1)}$ are the CPU registers. The number and use of these register vary considerably from CPU to CPU. These registers include general purpose registers and special purpose registers such as stack pointer, index registers and pointers.

There are two options provided for A input of the ALU. The multiplexer (MUX) is used to select one of the two inputs. It selects either output of Y register or a constant number as an A input for the ALU according to the status of the select input. It selects

output of Y when select input is 1 (select Y) and it selects a constant number when select input is 0 (select C) as an input A for the multiplexer. The constant number is used to increment the contents of program counter.

For the execution of various instructions CPU has to perform one or more of the following basic operations :

- Transfer a word of data from one CPU register to the another or to the ALU.
- Perform the arithmetic or logic operations on the data from the CPU registers and store the result in a CPU register.
- Fetch a word of data from specified memory location and load them into a CPU register.
- Store a word of data from a CPU register into a specified memory location.

Let us see how these basic operations are implemented using single bus organisation of processor.

3.2.1 Register Transfer

In Fig. 3.2.2, the data transfer between registers and common bus is shown by a line with arrow heads. But in actual practice each register has input and output gating and these gates are controlled by corresponding control signals. This is illustrated in Fig. 3.2.3. As shown in Fig. 3.2.3, control signals. $R_{i\ in}$ and $R_{i\ out}$ controls the input and output gating of register R_i. When $R_{i\ in}$ is set to 1, the data available on the common data bus is loaded into register R_i. Similarly, when $R_{i\ out}$ is set to 1, the contents of register R_i are placed on the common data bus. The signals $R_{i\ in}$ and $R_{i\ out}$ are commonly known as input enable and output enable signals of registers, respectively.

The input and output gates are nothing but the electronic switches which can be controlled by the

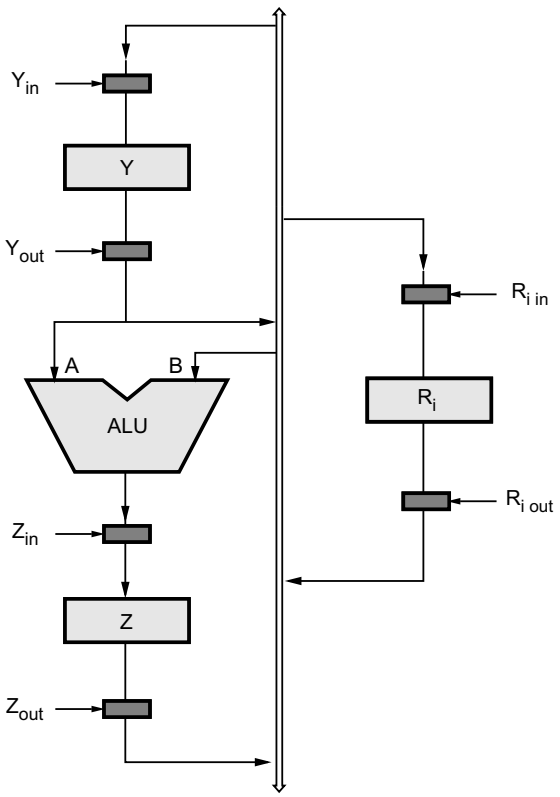


Fig. 3.2.3 Input and output gating for the register

control signals. When signal is 1, the switch is ON and when signal is 0, the switch is OFF.

The Fig. 3.2.4 shows the implementation of input and output gates of a 4-bit register.

Now consider that we have transfer data from register R_1 to R_2. This can be accomplished as follows :

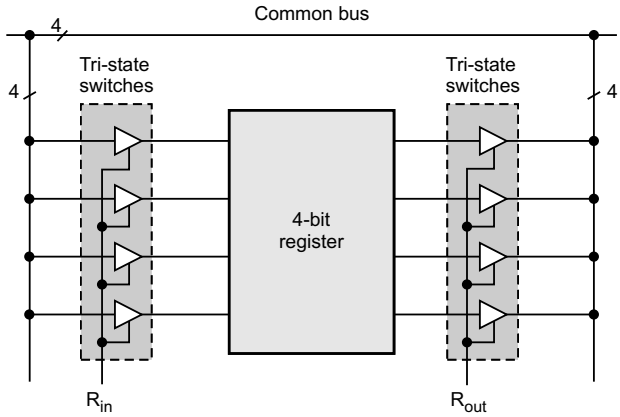


Fig. 3.2.4

- Activate the output enable signal of R_1, $R_{1\ out}$ = 1. This places the contents of R_1 on the common bus.
- Activate the input enable signal of R_2, $R_{2\ out}$ = 1. This loads data from the common bus into the register R_2.

All operations and data transfers within the processor take place in synchronisation with the clock signal. The control signals which controls a particular transfer are activated either at the rising edge or at the falling edge of the clock cycle.

The Fig. 3.2.5 shows the implementation of one bit register. The edge triggered D flip-flop which stores the one-bit data is connected to the common bus through tri-state switches. Input D is connected through input tri-state switch and output Q is connected through output tri-state switch. The control signal R_{in} enables the input tri-state switch and the data from common bus is loaded into the D flip-flip in synchronisation with clock input when R_{in} is active. This is implemented using AND gate as shown in the Fig. 3.2.5. The control signal R_{out} is activated to load data from Q output of the D flip-flop on to the common bus by enabling the output tri-state switch.

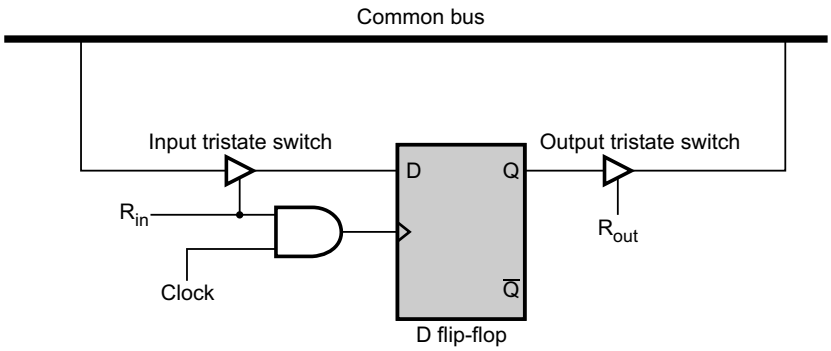


Fig. 3.2.5 One-bit register

3.2.2 Performing an Arithmetic or Logic Operation

ALU performs arithmetic and logic operations. It is a combinational circuit that has no internal memory. The ALU shown in Fig. 3.2.3 has two inputs A and B, and one output. Its A input gets the operand from the output of the multiplexer and its B input gets the operand directly from the bus. The result produced by the ALU is stored temporarily in register Z. Let us find the sequence of operations required to subtract the contents of register R_2 from register R_1 and store the result in register R_3. The sequence can be given as follows :

1. $R_{1\ out}$, Y_{in} 2. $R_{2\ out}$, Select Y, Sub, Z_{in} 3. Z_{out}, $R_{3\ in}$

Step 1 : In this step contents from register R_1 are loaded into register Y.

Step 2 : In this step contents from Y and from register R_2 are applied to the A and B inputs of ALU, respectively, subtraction is performed and result is stored in the Z register.

Step 3 : The contents of Z-register (result) is stored in the R_3 register.

3.2.3 Fetching a Word from Memory

To fetch a word of data from memory, the processor gives the address of the memory location where the data is stored on the address bus and activates the read operation. The processor loads the required address in MAR, whose output is connected to the address lines of the memory bus. At the same time processor sends the read signal of memory control bus to indicate the read operation. When the requested data is received from the memory it is stored into the MDR, from where it can be transferred to other processor registers.

The Fig. 3.2.6 shows the interface between processor and memory through MAR and MDR registers.

As shown in the Fig. 3.2.6, the MDR register has four control signals. Signals MDR_{in} and MDR_{out} control the connection to the internal processor data bus and signals Read and Write control the connection to the memory data bus. The MAR register has one control signal. The signal MAR_{in} controls the connection to the internal processor address bus. Control signals read and write from the processor controls the operation read and write, respectively.

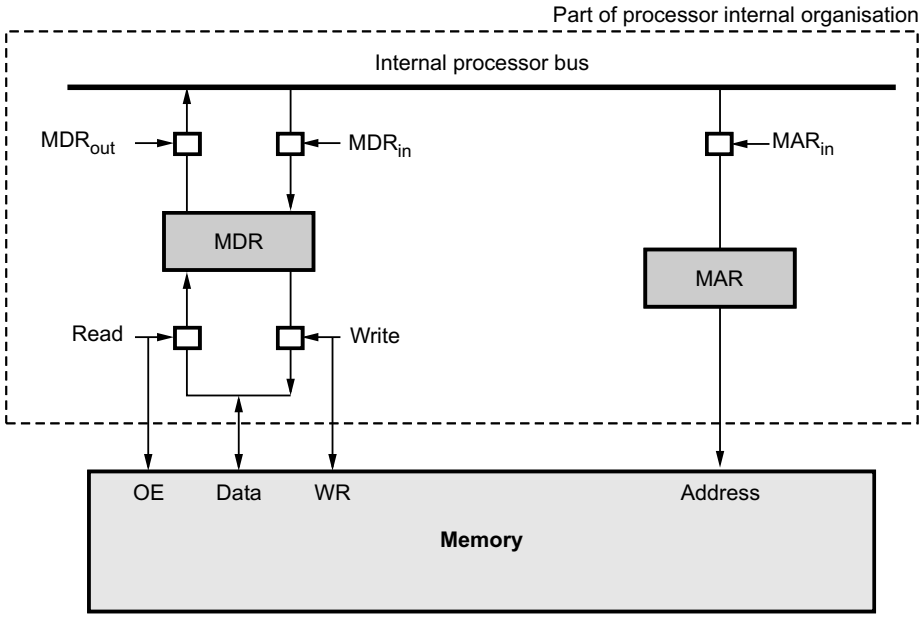


Fig. 3.2.6 Data, address and control signals for data transfer between memory and processor

Let us assume that the address of the memory word to be read is in register R_2 and we have to transfer data word from that location to the register R_3. This operation can be indicated by instruction MOVE R_3 , (R_2). The actions needed to execute this instruction are as follows :

1. MAR $\leftarrow$ [R_2]
2. Activate the control signal to perform the Read operation. If memory is slow, activate wait for memory function complete (WFC).
3. Load MDR from the memory bus.
4. $R_3 \leftarrow$ [MDR]

Let us see the various control signals which are necessary to activate to perform given actions in each step.

1. $R_{2\ out}$, MAR_{in} , Read
2. WMFC
3. MDR_{out} , $R_{3\ in}$

The Fig. 3.2.7 shows the timing diagram of a memory read operation.

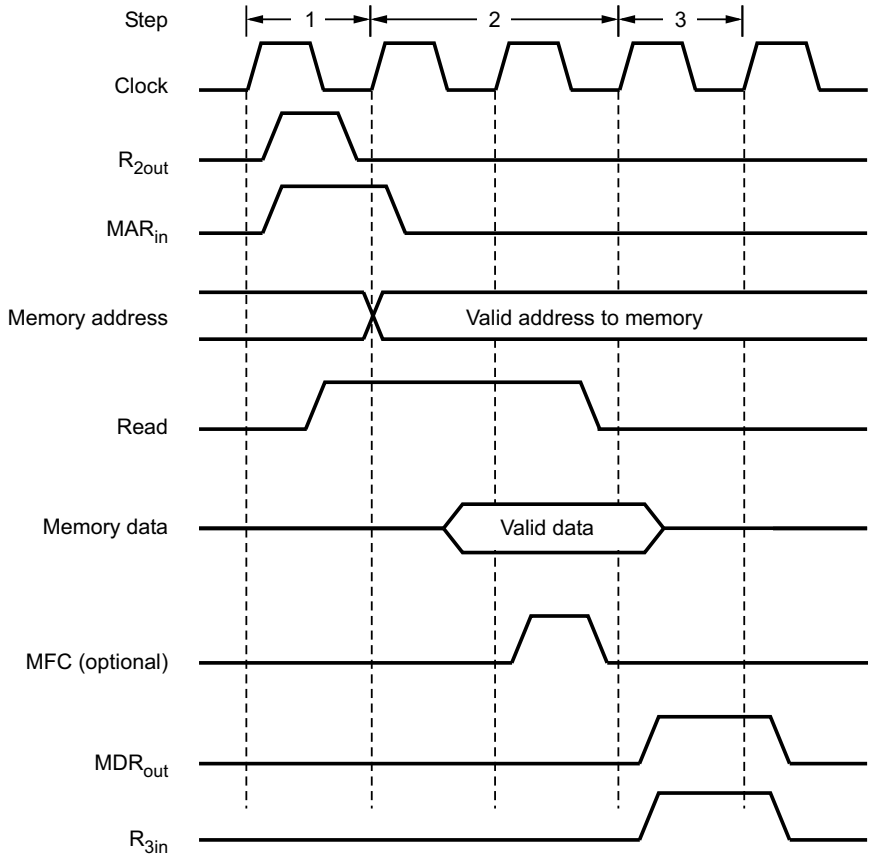


Fig. 3.2.7 Timing diagram for MOVE R_3, (R_2) instruction (memory read operation)

Note : 1. In above timing diagram R_{2out} and R_{3in} signals are shown as per the instruction. If instruction specifies any other registers, only control signals corresponding to specified register activate, rest of timing diagram will remain same.

2. In case of slow memory, the processor has to wait until the completion of memory read operation before going to perform next operation. In such cases control signal called Memory-Function Completed (MFC) is used. With MFC signal the step 2 becomes MAR_{out}, MDR_{in}, Read, WMFC (Wait for memory function complete).

3.2.4 Storing a Word in Memory

To write a word of data into a memory location processor has to load the address of the desired memory location in the MAR, load the data to be written in memory, in MDR and activate write operation. Let us assume that we have to execute instruction MOVE (R_2), R_1. This instruction copies the contents of register R_1 into the memory whose location is specified by the contents of register R_2. The actions needed to execute this instruction are as follow.

1. MAR $\leftarrow [R_2]$
2. MDR $\leftarrow [R_1]$
3. Activate the control signal to perform the write operation. If memory is slow, wait for memory function complete (WMFC).

Let us see the various control signals which are necessary to activate to perform given actions in each step.

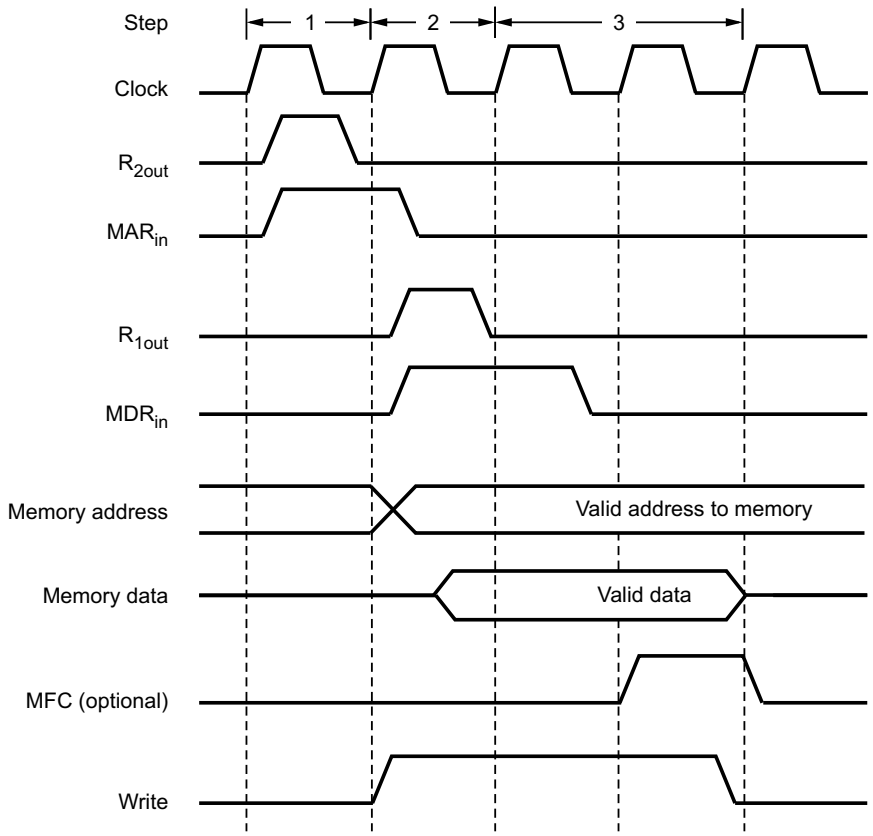


Fig. 3.2.8 Timing diagram for MOVE (R_2), R_1 instruction (Memory write operation)

1. R_{2out}, MAR_{in} 2. R_{1out}, MDR_{in}, Write 3. WMFC

The Fig. 3.2.8 shows the timing diagram of a memory write operation.

Note : In timing diagram signals R_{2out} and R_{1out} are shown as per the instruction. If instruction specifies any other registers, only control signals corresponding to specified register activate, rest of the timing diagram will remain same.

3.2.5 Execution of a Complete Instruction

Let us find the complete control sequence for execution of the instruction Add R_1, (R_2) for the single bus processor. This instruction adds the contents of register R_1 and the contents of memory location specified by register R_2 and store result in the register R_1. To execute bus instruction it is necessary to perform following actions.

1. Fetch the instruction.
2. Fetch the operand from memory location pointed by R_2.
3. Perform the addition.
4. Store the result in R_1.

The sequence of control steps required to perform these operations for the single bus architecture of Fig. 3.2.2, are as follows :

1. PC_{out}, MAR_{in}, Read, Y_{in}, Select C, Add, Z_{in}.
2. Z_{out}, PC_{in}, WMFC
3. MDR_{out}, IR_{in}
4. R_{2out}, MAR_{in}, Read
5. R_{1out}, Y_{in}, WMFC
6. MDR_{out}, Select Y, Add, Z_{in}
7. Z_{out}, $R_{1\ in}$, END

In step 1, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and activating Read signal. PC contents are also loaded into register Y and added with constant number by activating select C input of multiplexer and add input of the ALU. By activating Z_{in} signal result is stored in the register Z.

In step 2 the contents of register Z are transferred to PC register by activating Z_{out} and PC_{in} signal. This completes the PC increment operation and PC will now point to next instruction. After receiving WMFC signal, the contents of specified location are available in MDR register.

In step 3 contents of MDR register are transferred to the instruction register (IR) of the processor. The step 1 through 3 constitute the instruction fetch phase. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This

enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase.

In step 4, the contents of register R_2 are transferred to register MAR by activating R_{2out} and MAR_{in} signals and Read signal is activated.

In step 5, the contents of register R_1 are transferred to register Y by activating R_{1out} and Y_{in} signals. After receiving WMFC signal, the contents of specified location are available in MDR register.

In step 6, MDR_{out}, select Y, Add and Z_{in} signals are activated to perform addition of contents of register Y and the contents of MDR. The result is stored in the register Z.

In step 7, the contents of register Z are transferred to register R_1 by activating Z_{out} and R_{1in} signals. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

3.2.6 Branch Instruction

The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address. The branch target address is usually obtained by adding the offset in the contents of PC. The offset is specified within the instruction. The control sequence for unconditional branch instruction is as follows :

1. PC_{out}, MAR_{in}, Read, Y_{in} ,. SelectC, Add, Z_{in}
2. Z_{out}, PC_{in}, WMFC
3. MDR_{out}, IR_{in}
4. Offset_field_Of_IR_{out}, SelectY, Add, Z_{in}
5. Z_{out}, PC_{in}, End

The first three steps, step 1 through step 3 are same as in the previous example. They constitute the opcode fetch operation. In step 4 the contents of PC and the offset field of IR register are added and result is saved in register Z by activating corresponding signals.

In step 5, the contents of register Z are transferred to PC by activating Z_{out} and PC in signals.

In case of conditional branch instruction the status of the condition code specified in the instruction is checked. If the status specified within the instruction matches with the current status of condition code the branch target address is loaded in the PC by adding the offset specified in the instruction to the contents of PC otherwise processor fetches the next instruction in the sequence.

3.2.7 Multiple Bus Organization

So far we have discussed the basic operations of processor using single bus architecture of the processor. However, because of single bus only one data word can be transferred over the bus in a clock cycle. This increases the steps required to complete the execution of the instruction. To reduce the number of steps needed to execute instructions, most commercial processors provide multiple internal paths that enable several transfer to take place in parallel.

The Fig. 3.2.9 shows a three-bus structure of the processor. (See Fig. 3.2.9 on next page.) Here, three buses are used to connect registers and the ALU of the processor. In the figure all general purpose registers are shown by a single block called **register file**. The register file shown in Fig. 3.2.9 has three ports : one input port and two output ports. So it is possible to access data of three register in one clock cycle; the value can be loaded in one register from bus C and data from two register can be accessed to bus A and bus B, respectively. Buses A and B are used to transfer the source operands to the A and B inputs of the ALU. After performing arithmetic or logic operation result is transferred to the destination operand over bus C.

To increment the contents of PC after execution of each instruction to fetch the next instruction, separate unit is provided. This unit is known as **incrementer**. Incrementer increments the contents of PC accordingly to the length of the instruction so that it can point to next instruction in the sequence. The incrementer eliminates the need of multiplexer connected at the A input of ALU.

Let us consider the execution of instruction, Add R_1, R_2, R_3. This instruction adds the contents of registers R_2 and the contents of register R_3 and stores the result in R_1. With three-bus organisation control steps required for execution of instruction Add R_1, R_2, R_3 are as follows :

1. PC_{out}, R = B, $MAR_{in,}$ Read, IncPC
2. WMFC
3. MDR_{out}, IR_{in}, R = B
4. R_{2out}, R_{3out}, Add, R_{1in}, End

In step 1, the contents of PC are transferred to MAR through Bus B to start the Read operation and simultaneously PC is incremented to point the next instruction . In step 2, the processor waits for MFC. In step 3, the instruction code is transferred from MDR to IR register. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control signals for step 4, which constitute the execution phase.

In step 4, two operands from register R_2 and register R_3 are made available at A and B inputs of ALU through BUS A and BUS B. These two inputs are added by activation of **Add** signal and result is stored in R_1 through Bus C.

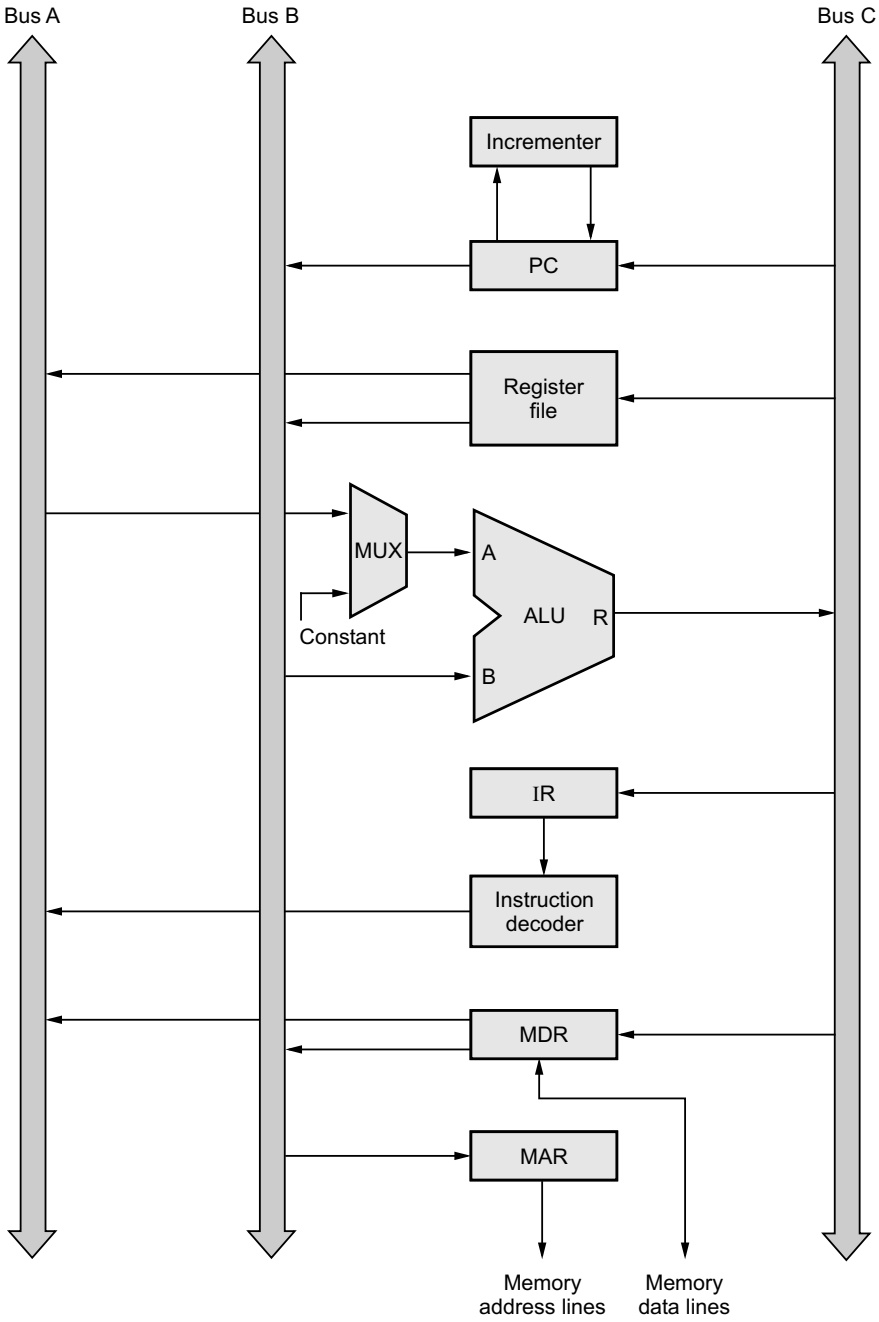


Fig. 3.2.9 Three-bus organisation of processor

From the above example we can realise that by providing more data paths it is possible to reduce the number of clock cycles needed to execute an instruction, significantly.

Example 3.2.1 *Explain how the instruction A = B – C gets executed in a system in detail.* **Dec.-08, Marks 8**

Solution : Let us consider that registers A, B and C are available in the three-bus organization. With three-bus organization control steps required for execution of instruction A = B – C are as follows :

1. PC_{out}, R = B, MAR_{in}, Read, IncPC
2. WMFC
3. MDR_{out}, IR_{in}, R = B
4. B_{out}, C_{out}, Sub, A_{in}, End

In step 1, the contents of PC are transferred to MAR through Bus B to start the Read operation and simultaneously PC is incremented to point the next instruction . In step 2, the processor waits for MFC. In step 3, the instruction code is transferred from MDR to IR register. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control signals for step 4, which constitute the execution phase.

In step 4, two operands from register B and register C are made available at A and B inputs of ALU through BUS A and BUS B. The operand C is subtracted from operand B by activation of **sub** signal and result is stored in register A through bus C.

Review Questions

1. *Draw the diagram of the single bus organization of the data path inside a processor. Explain the above regarding execution.* **Dec.-10, Marks 8**
2. *Explain how gating is achieved in registers.* **Dec.-08, Marks 6**
3. *Give the sequence of control signals to be generated to fetch an instruction from memory in a single-bus organization.* **May-06, 08, Marks 4**
4. *Why is the Wait-for-Memory-Function-Completed step needed when reading from or writing to the main memory ?* **May-06, 08, Marks 2**
5. *Explain how the processor is interfaced with the memory with neat block diagram and explain how they communicate.* **Dec.-07, Marks 10**
6. *Illustrate memory read operation.* **May-07, Marks 7**
7. *Illustrate memory write operation.* **May-07, Marks 7**
8. *Explain the execution of an instruction with diagram.* **May-07, 08, Marks 8**
9. *Explain multiple bus organization in detail.* **May-07, Marks 6**
10. *Write the sequence of control steps required for three bus structure for the following instruction : Add R4, R5, R6.* **May-08, Marks 2**

3.3 Hardwired Control

May-07, 10, 11, Dec.-03, 06, 07, 08, 11

In the hardwired control, the control units use fixed logic circuits to interpret instructions and generate control signals from them. The fixed logic circuits use contents of the control step counter, contents of the instruction register, contents of the condition code flag and the external input signals such as MFC and interrupt requests to generate control signals. Fig. 3.3.1 shows the typical hardwired control unit. Here, the fixed logic circuit block includes combinational circuit (decoder and encoder) that generates the required control outputs, depending on the state of all its inputs. By separating the decoding and encoding functions, we can draw more detail block diagram for hardwired control unit as shown in the Fig. 3.3.2.

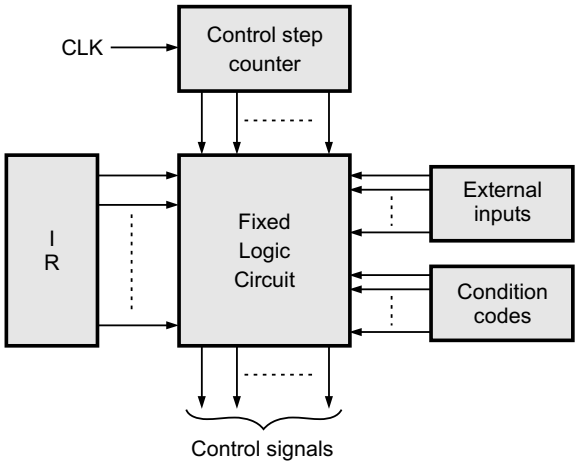


Fig. 3.3.1 Typical hardwired control unit

The instruction decoder decodes the instruction loaded in the IR. If IR is an 8-bit register then instruction decoder generates 2^8, i.e. 256 lines; one for each instruction.

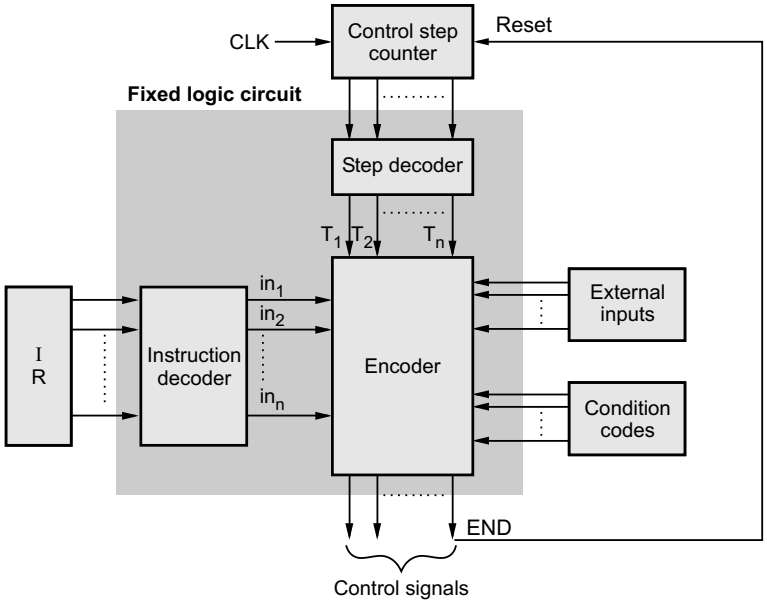
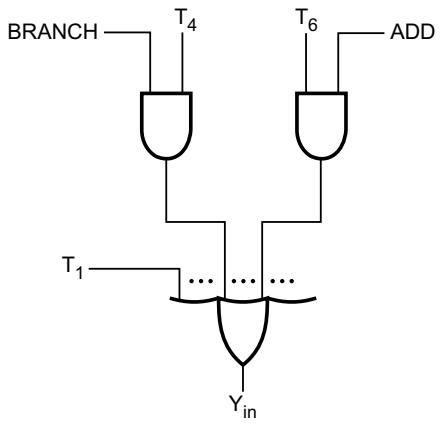


Fig. 3.3.2 Detail block diagram for hardwired control unit

According to code in the IR, only one line amongst all output lines of decoder goes high i.e., set to 1 and all other lines are set to 0. The step decoder provides a separate signal line for each step, or time slot, in a control sequence. The encoder gets in the input from instruction decoder, step decoder, external inputs and condition codes. It uses all these inputs to generate the individual control signals. After execution of each instruction end signal is generated which resets control step counter and make it ready for generation of control step for next instruction.

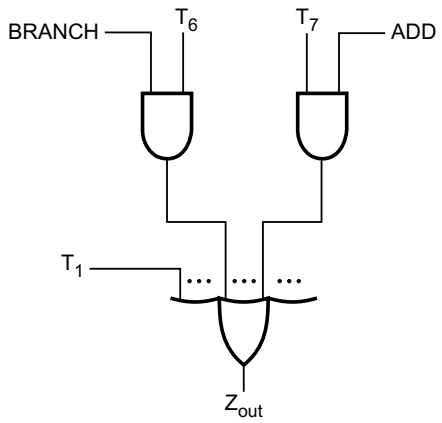
Let us see how the encoder generates signal for single bus processor organisation shown in Fig. 3.3.3 Y_{in}. The encoder circuit implements the following logic function to generate Y_{in}.

$$Y_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BRANCH + \ldots$$



Generation of the Y_{in} control signal

Fig. 3.3.3



Generation of the Z_{out} control signal

Fig. 3.3.4

The Y_{in} signal is asserted during time interval T_1 for all instructions, during T_6 for an ADD instruction, during T_4 for an unconditional BRANCH instruction and so on.

As another example, the logic function to generate Z_{out} signal can given by,

$$Z_{out} = T_2 + T_7 \cdot ADD + T_6 \cdot BRANCH + \ldots$$

The Z_{out} signal is asserted during time interval T_2 of all instructions, during T_7 for an ADD instruction, during T_6 for an unconditional branch instruction and so on.

The Fig. 3.3.3 and 3.3.4 show the hardware implementation of logic functions for Y_{in} and Z_{out} control signals.

Example 3.3.1 *Generate the logic circuit for the following function*

$$End = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot BRN + \ldots$$ **May-11, Marks 4**

Solution : The Fig. 3.3.5 shows the circuit that generates the End control signal from the logic function.

$$End = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot BRN + \ldots\ldots$$

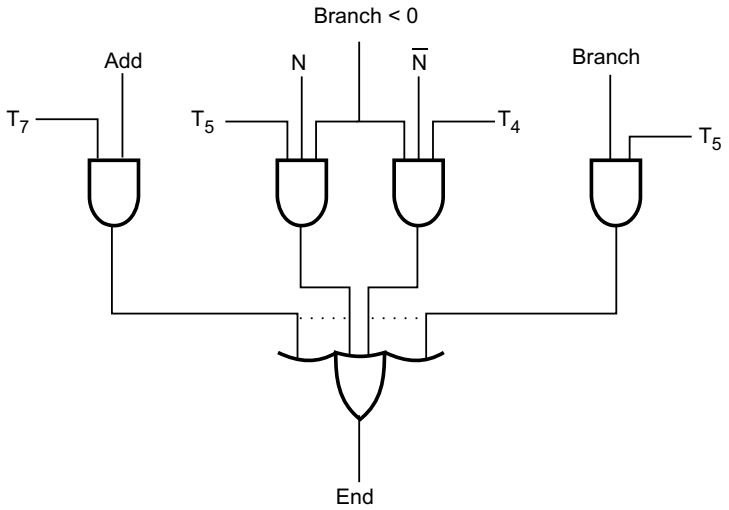


Fig. 3.3.5 Generation of the end control signal

Advantages of hardwired control unit

- Hardwired control unit is fast because control signals are generated by combinational circuits.
- The delay in generation of control signals depends upon the number of gates.
- It has greater chip area efficiency since its uses less area on-chip.

Disadvantages of hardwired control unit

- More the control signals required by CPU; more complex will be the design of control unit.
- Modifications in control signal are very difficult. That means it requires rearranging of wires in the hardware circuit.
- It is difficult to correct mistake in original design or adding new feature in existing design of control unit.

3.3.1 Design Methods of Hardwired Control Unit

There are four simplified and systematic methods for the design of hardwired controllers.

- **State-table Method :** It is the standard algorithmic approach to sequential circuit design.

- **Delay-element Method :** It is a heuristic method based on the use of clocked delay elements for control signal timing.
- **Sequence-counter Method :** It uses counters for timing purposes.
- **PLA Method :** It uses programmable logic array.

Let us consider the 2's complement multiplier control circuit for illustration of above designing methods. The hardwired control unit for 2's complement multiplication is already introduced. Here, the same block diagram (Fig. 3.3.6) is redrawn with control signals. The operation controlled by each control signal is listed in Table 3.3.1.

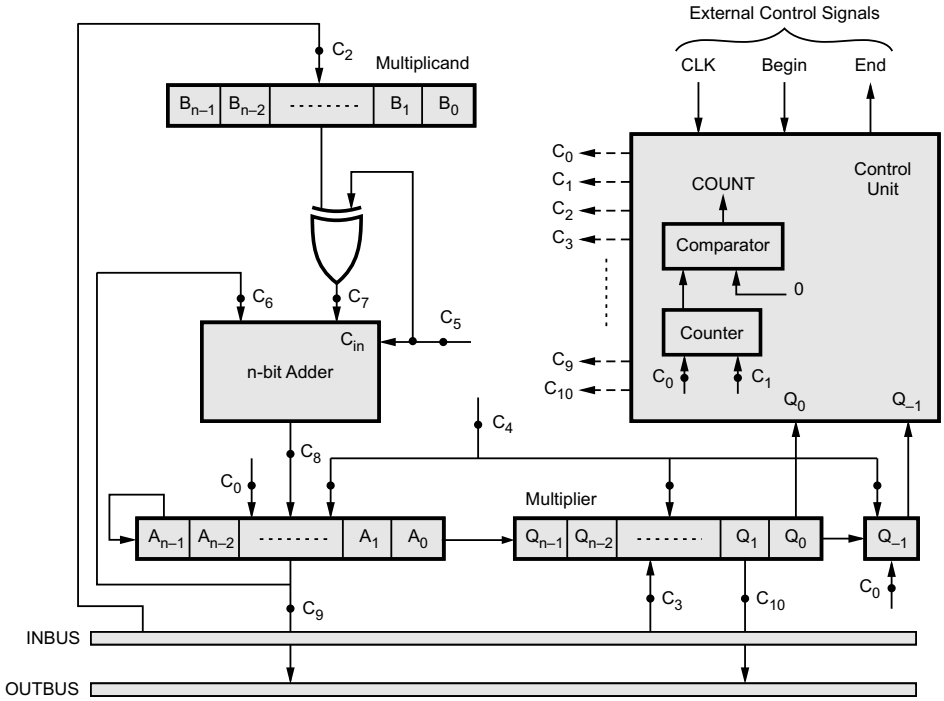


Fig. 3.3.6 Multiplier control circuit with control signals

Control signal	Operation controlled
C_0	Clear A, Q_{-1} and count ← n
C_1	Decrement count
C_2	Transfer word on INBUS to B
C_3	Transfer word on INBUS to Q
C_4	Shift right register A, Q and Q_{-1}
C_5	2's complement multiplicand

C_6	Transfer A to left input of adder
C_7	Transfer B to right input of adder
C_8	Transfer adder output to A
C_9	Transfer A to OUTBUS
C_{10}	Transfer Q to OUTBUS

Table 3.3.1 Control signals for the two's complement multiplier

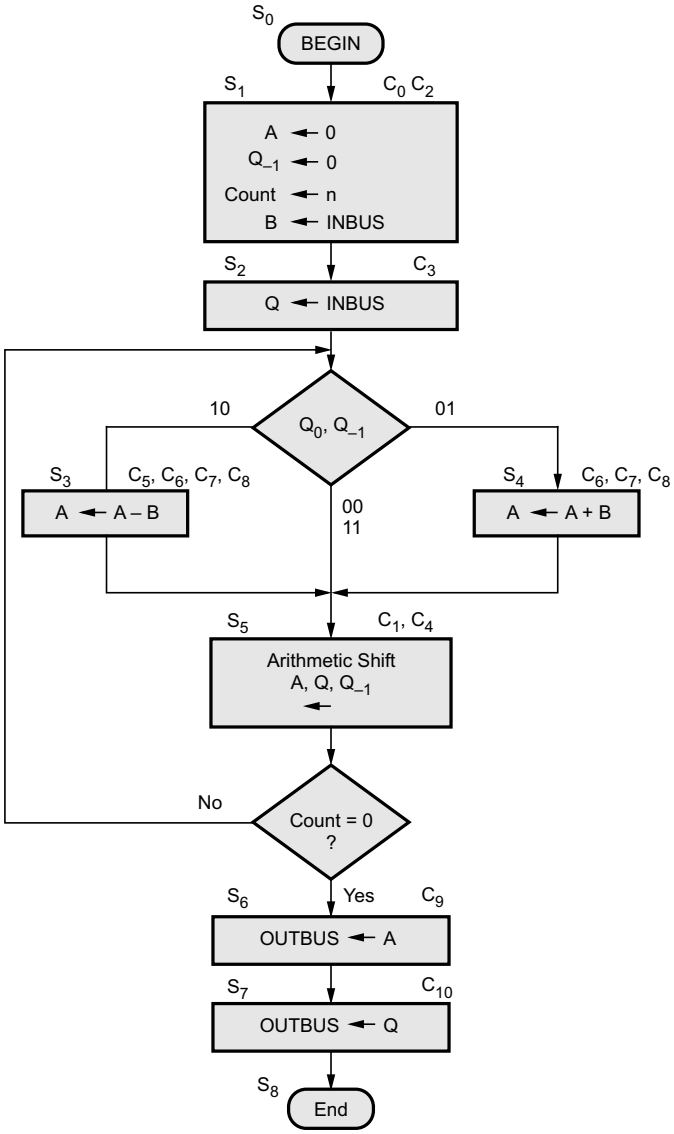


Fig. 3.3.7 Flowchart for 2's complement multiplication

In general, a control point is associated with each distinct action/operation. A common control signal is used for operation which can take place simultaneously. For example register A and Q_{-1} can be reset and COUNT can be loaded simultaneously. Hence only one control signal C_0 is assigned for three operations. Control signal C_1 is used to decrement the iteration count. Control signals C_2 and C_3 transfer a data word from the input bus INBUS to registers B and Q, respectively and are shown in the corresponding data paths of Fig. 3.3.7. In case of subtraction, control signal C_5 does the 2's complement conversion of the multiplicand (Ex-OR gate with one input logic 1 inverts the input to get the 1's complement of multiplicand and after addition of Cin (1) we get the 2's complement of the multiplicand). Control signals C_6 and C_7 transfer a data from A and B registers to the left and right inputs of adder, respectively. Control signal C_8 is used to transfer adder output to A register. To transfer final result from A and Q registers on the output bus, OUTBUS, control signals C_9 and C_{10} are used. The control signal COUNT is set to 0 when counter has non-zero count; otherwise it is set to logic 1. BEGIN and COUNT signals, the rightmost bit Q_0 of the Q register and register serve as the primary inputs of the control unit. On the completion of multiplication process control unit activates END signal.

The 2's complement multiplication algorithm. The Fig. 3.3.7 shows flow chart of it indicates at what instant the control signals from the Fig. 3.3.6 are activated. With this information in hand, we now apply the three design techniques for hardwired implementation of multiplier control unit.

3.3.1.1 State-table Method

In this approach, it is necessary to construct a state table for the control unit. We can associate a state with every micro-operation block, as shown in Fig. 3.3.7, giving nine states labeled S_0 through S_8. S_0 state represents the idle or waiting state of the control unit. As mentioned earlier, there are four primary input signals BEGIN, COUNT, Q_0 and Q_{-1}, hence there are sixteen possible input combinations. Table 3.3.2 shows a state table for the control unit which is directly derived from the flowchart shown in Fig. 3.3.7. Each entry indicates the next state followed by a list of the control signals that are activated. For example, for S_1, the next state is S_2 and it is reached by activating control signals C_0 and C_2. Thus the state table entry is S_2, C_0, C_2. Certain state-and-input-signal combinations should not occur during normal operation, so the corresponding table entries are left blank. For example, the BEGIN signal should assume the 1 value only when the control unit is in the idle state S_0. Similarly, COUNT (which becomes 1 when counter count = 0) is never 1 in state S_2, since counter is loaded with count n in the preceding state S_1.

Inputs				States								
BEGIN	COUNT	Q_0	Q_{-1}	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
0	0	0	0	S_0	S_2, C_0 C_2	S_5, C_3			S_5, C_1, C_4			
0	0	0	1	S_0	S_2, C_0 C_2	S_4, C_3		S_5, C_6 C_7, C_8	S_4, C_1, C_4			
0	0	1	0	S_0	S_2, C_0 C_2	S_3, C_3	S_5, C_5 C_6, C_7 C_8		S_3, C_1 C_4			
0	0	1	1	S_0	S_2, C_0 C_2	S_5, C_3			S_5, C_1 C_4			
0	1	0	0	S_0	S_2, C_0 C_2				S_6, C_1 C_4	S_7, C_9	S_8, C_{10}	S_0, END
0	1	0	1	S_0	S_2, C_0 C_2				S_6, C_1 C_4	S_7, C_9	S_8, C_{10}	S_0, END
0	1	1	0	S_0	S_2, C_0 C_2				S_6, C_1, C_4	S_7, C_9	S_8, C_{10}	S_0, END
0	1	1	1	S_0	S_2, C_0 C_2				S_6, C_1 C_4	S_7, C_9	S_8, C_{10}	S_0, END
1	0	0	0	S_1								
1	0	0	1	S_1								
1	0	1	0	S_1								
1	0	1	1	S_1								
1	1	0	0	S_1								
1	1	0	1	S_1								
1	1	1	0	S_1								
1	1	1	1	S_1								

Table 3.3.2 State table for multiplier control unit

Once the state table is ready, it can be implemented using JK flip-flops.

3.3.1.2 Delay Element Method

The control signals or groups of control signals from the control unit are activated in a proper sequence. There is a specific time delay between activation of two conjugative control signals or groups of control signals. A sequence of delay elements can be used to generate control signals one after the another. To ensure synchronous operation, the delay elements are implemented by D flip-flops and controlled by a common clock signal.

A control unit using delay elements can be constructed directly from flowchart that specifies required control signal sequences. Before going to see the implementation of multiplier control unit we see the simple rules to drive control circuit from the flowchart.

Rule 1 : Each sequence of two successive micro-operations require a delay element. The signals that activate the control lines are taken directly from the input and output lines of the delay element as shown in the Fig. 3.3.8 (a). The signals that are intended to activate same control line are logically ORed to get one common output signal as shown in Fig. 3.3.8 (b).

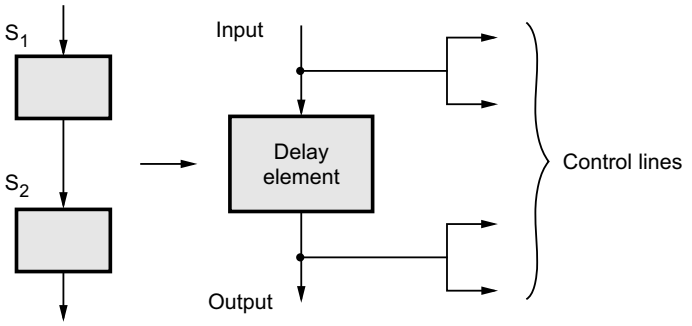


Fig. 3.3.8 (a)

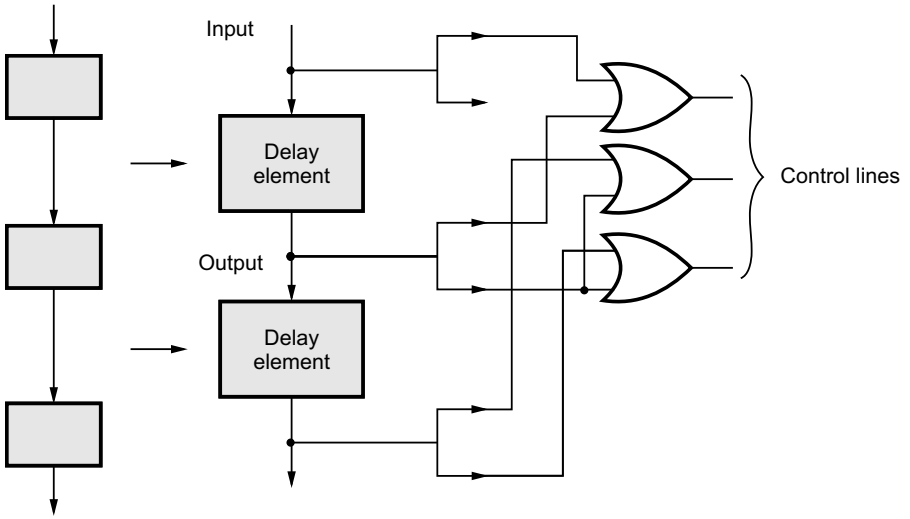


Fig. 3.3.8 (b)

Rule 2 : n lines in the flowchart merge to a common line are transformed into input OR gate, as shown in Fig. 3.3.9.

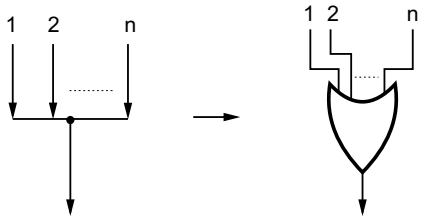


Fig. 3.3.9

Rule 3 : A decision box can be implemented by two AND gates, as shown in Fig. 3.3.10.

Let us see the implementation of multiplier control unit directly from the flowchart shown in Fig. 3.3.10 by using the transformation rules. The Fig. 3.3.11 shows resultant circuit. (Refer Fig. 3.3.11 on next page).

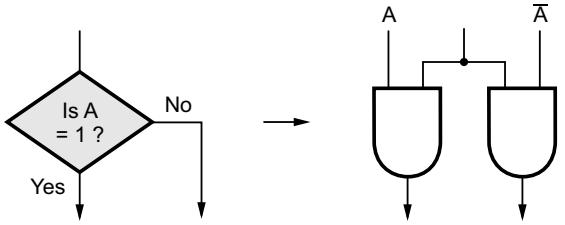


Fig. 3.3.10

3.3.1.3 Sequence Counter Method

Fig. 3.3.12 (a) shows a control unit based on a sequence counter and Fig. 3.3.12 (b) shows internal circuit of sequence counter. The sequence counter consists of modulo-k counter (counter which counts from 0 to k–1 states), 1/k decoder and input signals to control the operation of modulo-k counter. The output of the modulo-k counter is connected to the input of 1/k decoder. When the count enable input is connected to a clock source, the counter cycles continually through its k states, and decoder generates k pulse signals (ϕ_i) on its output lines. These are called as phase signals. Consecutive pulses of phase signals are separated by one clock period, as shown in the Fig. 3.3.12 (b). Two additional input lines (begin and end) and a flip-flop are provided for turning the counter on and off. A pulse on the begin line causes the counter to begin cycling through its states by logically connecting the count enable signal to the clock source. A pulse on the end line disconnects the clock and resets the counter.

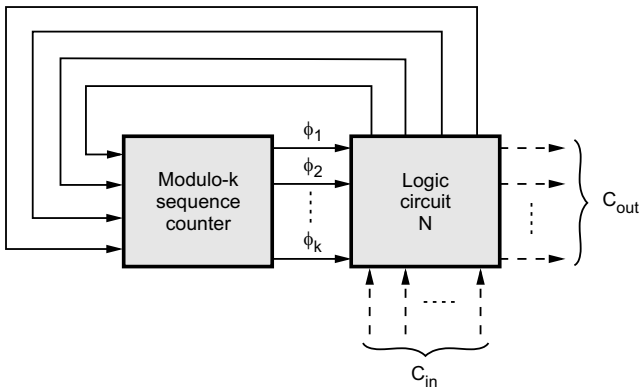


Fig. 3.3.12 (a) A control unit based on a sequence counter

The phase signals (ϕ_i) are used to activate some set of control lines in each microoperation of the instruction. As instruction changes, microoperations also change. For these changed microoperations, the phase signals must activate some other set of control lines. The set of control lines depend on certain control signals or condition variables applied to the control unit. To activate appropriate set of control lines, it is necessary to connect logic circuit to the modulo-k sequence counter, as shown in Fig. 3.3.12 (a). To get a clear idea now we design multiplier control unit using sequence counter. Look at the flowchart for 2's complement multiplication, shown in Fig. 3.3.13. It

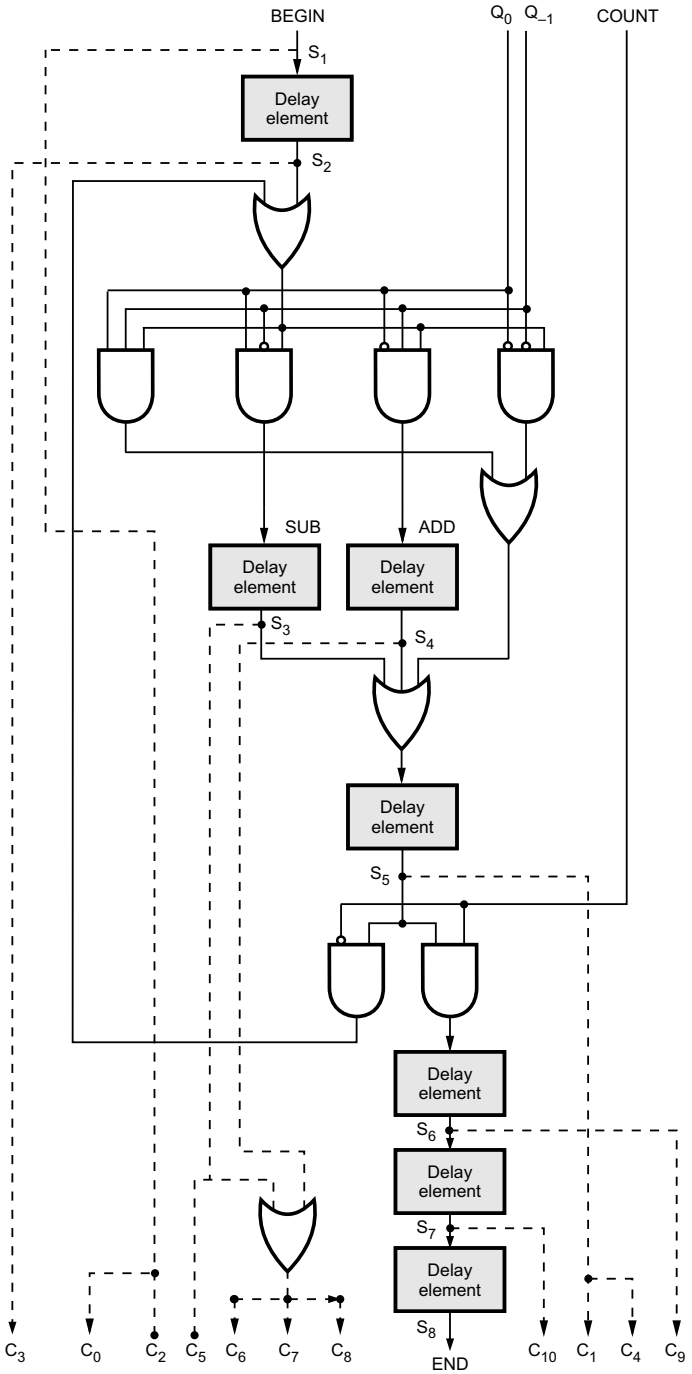
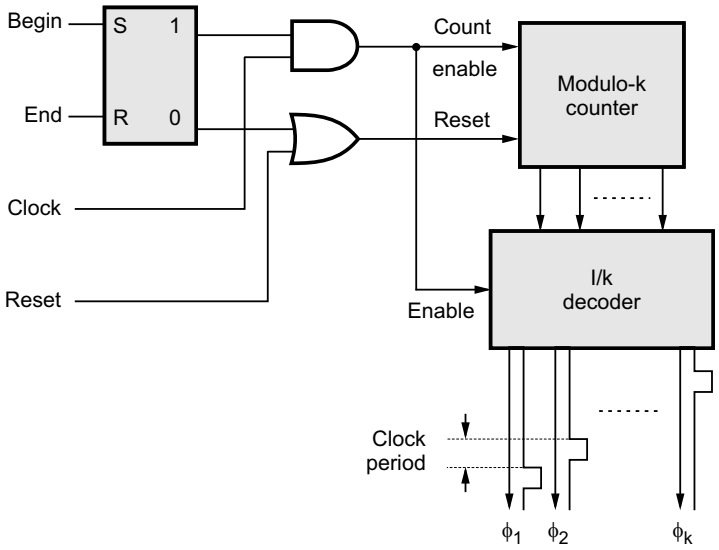
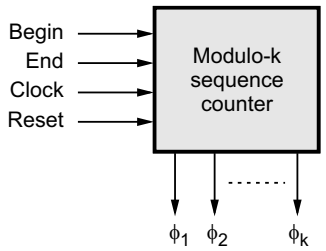


Fig. 3.3.11 Multiplier control unit using delay elements



Logic diagram



Symbol

Fig. 3.3.12 (b)

is redrawn with identification of different phases in it. The phase 1 does initial settings. It does this in two steps :

Step 1 : (A ← 0, Q_{-1} ← 0 , COUNT ← n and B ← INBUS) and

Step 2 : (Q ← INBUS).

Phase 2 is repeated n times and it also has two steps :

Step 1 : (A ← A – B or A ← A + B) and

Step 2 : (Arithmetic shift A, Q, Q_{-1} and COUNT ← COUNT – 1).

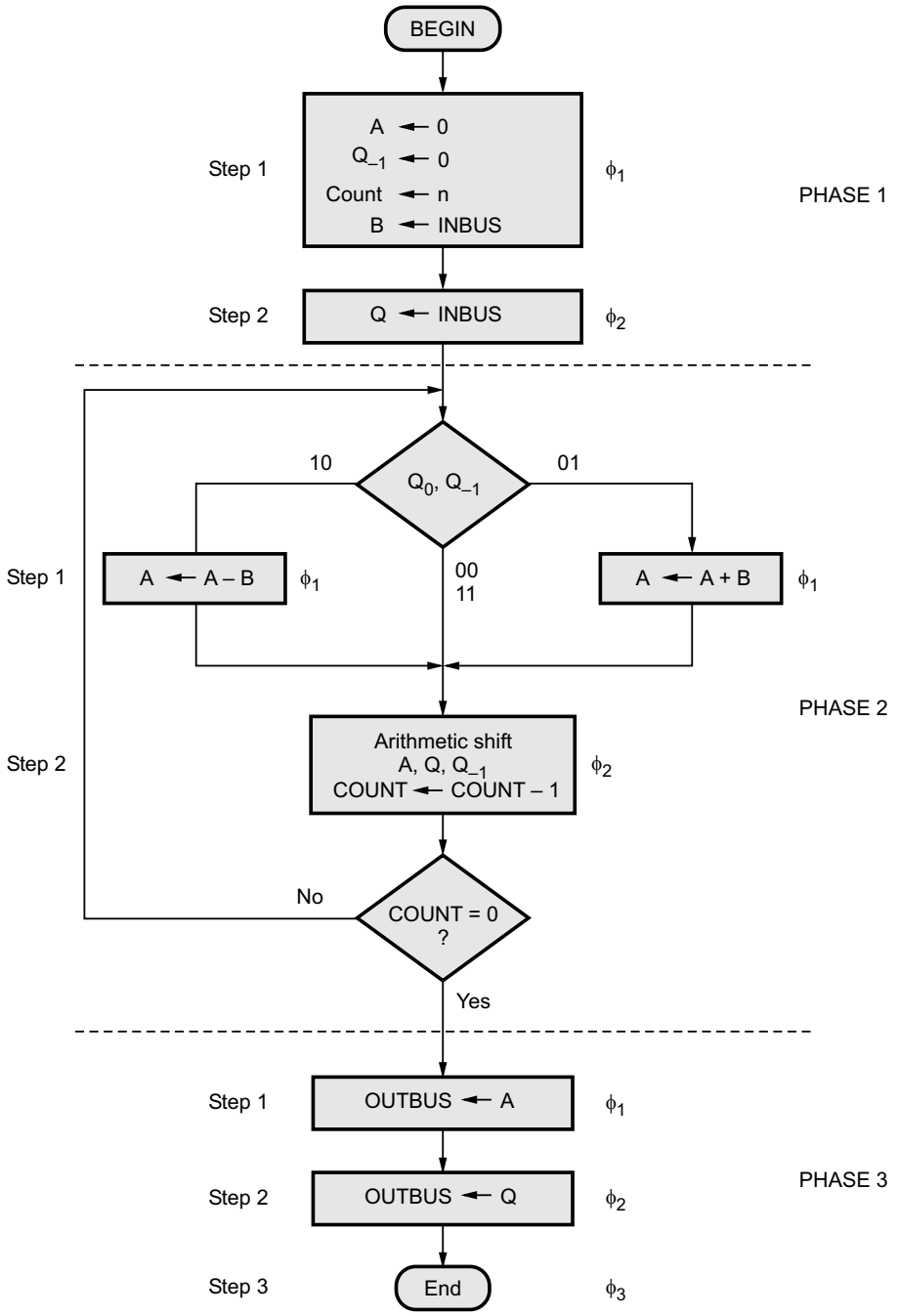


Fig. 3.3.13 Flowchart for 2's complement multiplication

Phase 3 involves three steps :

Step 1 : (OUTBUS ← A), **Step 2 :** (OUTBUS ← Q), and **Step 3 :** (Activate END).

In each step of particular phase, the corresponding phase signal is used to activate particular set of control lines, as shown in the Fig. 3.3.14 (See Fig. 3.3.14 on next page).

Begin signal sets the output of F/F_1 to logic 1, activating the phase 1. In phase 1 counter goes through steps 0 to 2 and activates ϕ_1, ϕ_2 and ϕ_3 signals in sequence. This action activates control signals as shown in Table 3.3.3.

Phase 1 Phase signal	Activated control lines	Enabled AND gates
ϕ_1	C_0 and C_2	1
ϕ_2	C_3	2
ϕ_3	Resets F/F_1 and sets F/F_2	3

Table 3.3.3

The phase signal resets F/F_1 and sets F/F_2 activating phase 2. In phase 2, phase signals to are activated again to activate control signals as shown in Table 3.3.4.

Phase 2 Phase signal	Activated control lines	Enabled AND gates
ϕ_1	In case of subtraction : C_5, C_6, C_7 and C_8	4
	In case of addition : C_6, C_7 and C_8	5
ϕ_2	C_1 and C_4	7
ϕ_3	Resets F/F_2 and sets F/F_3, if COUNT = 1	6

Table 3.3.4

The phase 2 is repeated until COUNT = 0. When COUNT signal goes high, phase signal resets F/F_2 and sets F/F_3 activating phase 3. In phase 3, phase signals are activated again to activate control signals as shown in the Table 3.3.5.

Phase 3 Phase signal	Activated control lines	Enabled AND gates
ϕ_1	C_9	8
ϕ_2	C_{10}	9
ϕ_3	END	10

Table 3.3.5

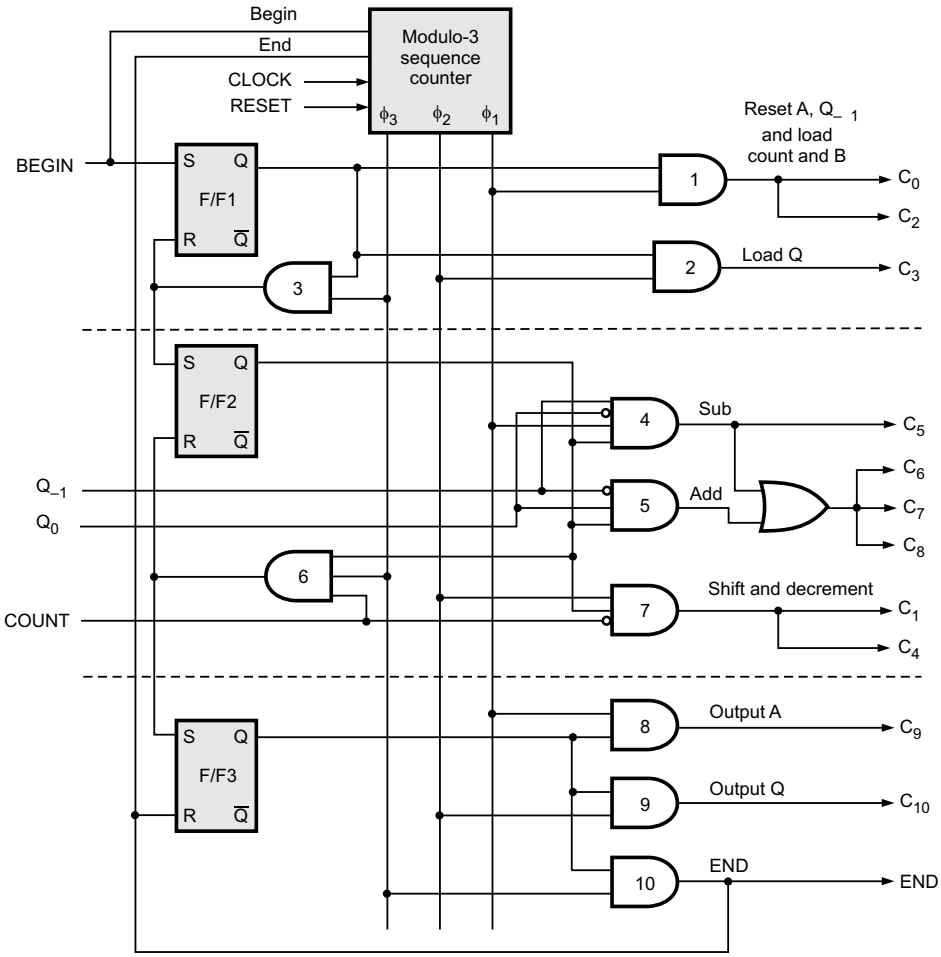


Fig. 3.3.14 Multiplier control unit using sequence counter

3.3.1.4 PLA Method

The control unit designed methods discussed so far are suitable only for small control units due to their size and complexity. In modern computer VLSI technology based circuitry is used for the design of control unit. One such structure, programmable logic array (PLA) is discussed in this section. A PLA consists of an array of AND gates followed by an array of OR gates, as shown in the Fig. 3.3.15.

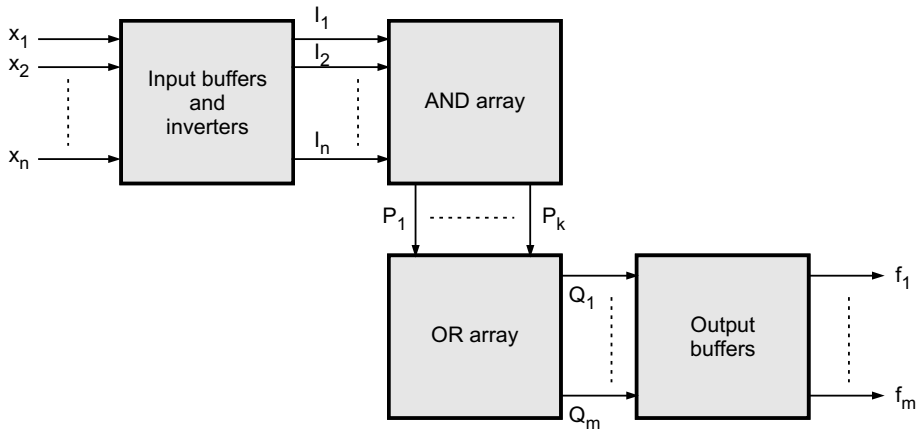


Fig. 3.3.15 Block diagram of PLA

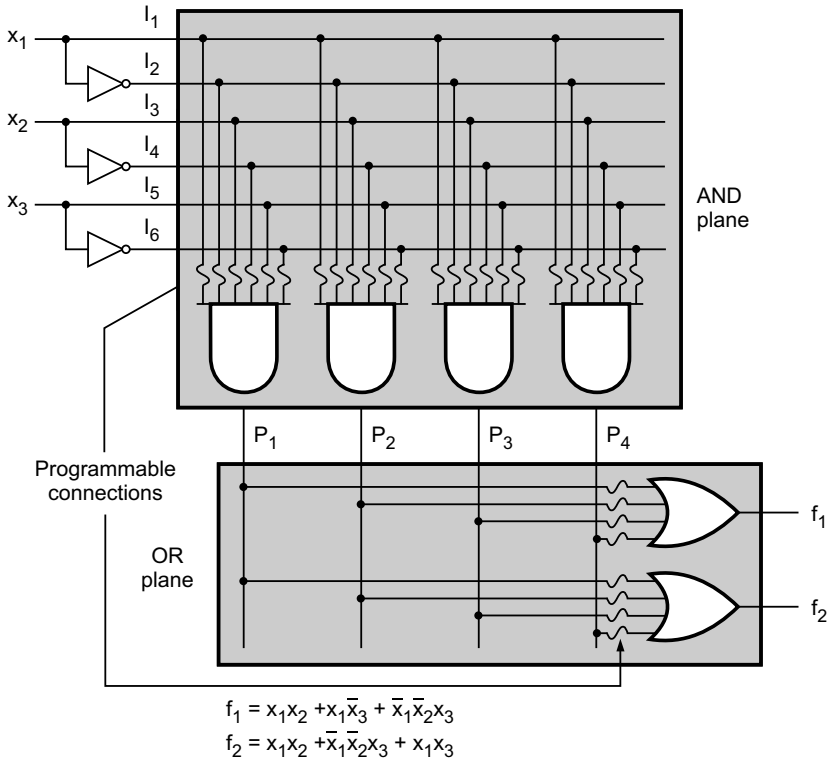


Fig. 3.3.16 Function structure of a PLA

Here, AND and OR arrays both can be programmed to implement combinational logic functions of several variables. Fig. 3.3.16 illustrates the functional structure of a PLA using a simple example. x_1, x_2 and x_3 are the input signals and f_1, f_2, the

combinations of the input signals are available as outputs. Using same logic, we can give instruction code, contents of control step counter, flags and condition codes as an input to the PLA and programmed AND and OR arrays to get outputs as control signals. This is illustrated in Fig. 3.3.17.

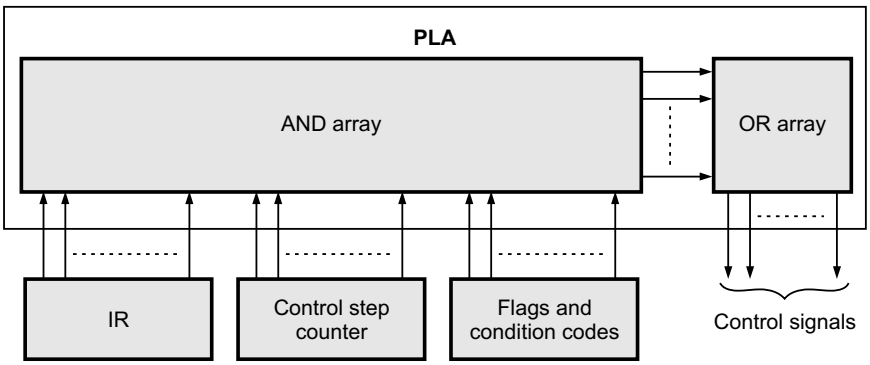


Fig. 3.3.17 Control unit implementation using PLA

3.3.2 A Complete Processor

The Fig. 3.3.18 shows the block diagram of complete processor. This block diagram consists of instruction unit, integer unit, floating-point unit, instruction cache, data cache, bus interface unit, main memory module and Input/Output module. The instruction unit fetches instructions from an instruction cache or from the main memory when the desired instructions are not available in the cache. The complete processor provides two processing units : Integer unit and floating-point unit. Integer unit to process integer data and floating-point to process floating-point data. These two units gets data from data cache. Use of separate caches for instructions and data is common practice in many processors today. Other processor use a common single cache for both instruction and data. The 80486 processor has 8-kbytes single cache for both instruction and data whereas the Pentium processor has two separate 8-kbytes caches for instruction and data. The processor provides bus interface unit to control the interface of processor to system bus, main memory module and input/output module.

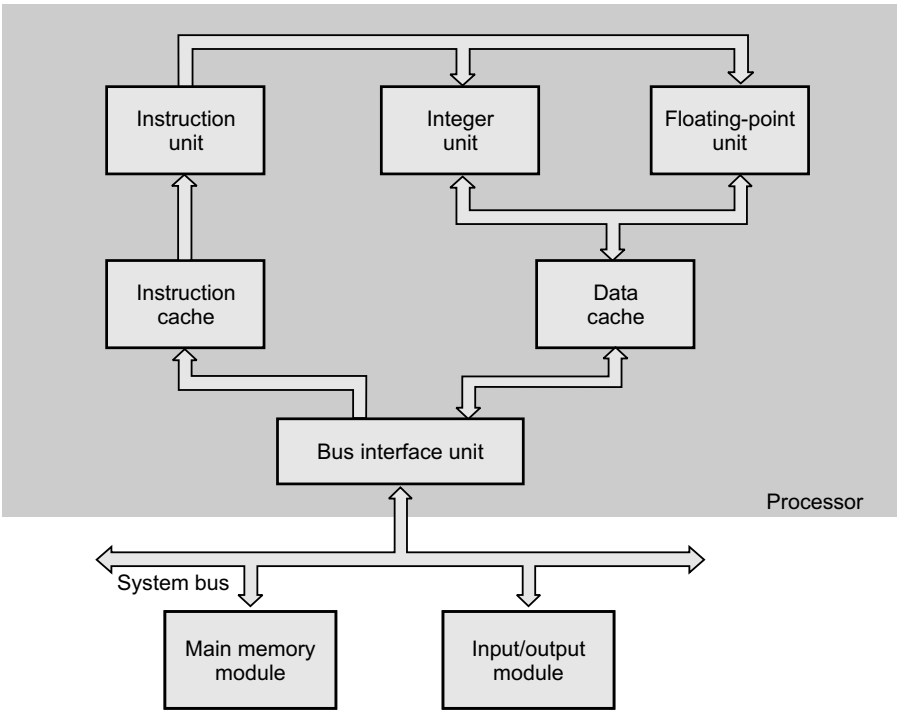


Fig. 3.3.18 Block diagram of a complete processor

3.3.3 CPU Control Unit

Consider the accumulator based CPU as shown in Fig. 3.3.19. It consists of a datapath unit DPU designed to execute the set of 10 basic single address instructions listed in Table 3.6. It is assumed that the instructions are of fixed length and they act on data words of the same fixed length, say 32 bits. The program control unit PCU is responsible for managing the control signals linking the PCU to the DPU, as well as the control signals between the CPU and the external memory M.

Instruction Type	Instruction	Comment
Data transfer	LD X	AC ← M(X) : Load data from location X in the memory into AC
	ST X	M(X) ← AC : Store contents of AC in location X of memory
	MOV DR, AC	DR ← AC : Copy contents of AC to DR
	MOV AC, DR	AC ← DR : Copy contents of DR to AC

Data processing	ADD	AC ← AC + DR : Add AC to DR and store result in AC
	SUB	AC ← AC – DR : Subtract DR from AC and store result in AC
	AND	AC ← AC and DR : AND DR to AC and store result in AC
	NOT	AC ← not AC : Complement contents of AC
Program control	BRA addr	PC ← M(addr) : Jump to instruction with address addr.
	BZ addr	If AC = 0 then : Jump to instruction with address PC ← M (addr) addr if AC = 0.

Table 3.3.6 Instruction set

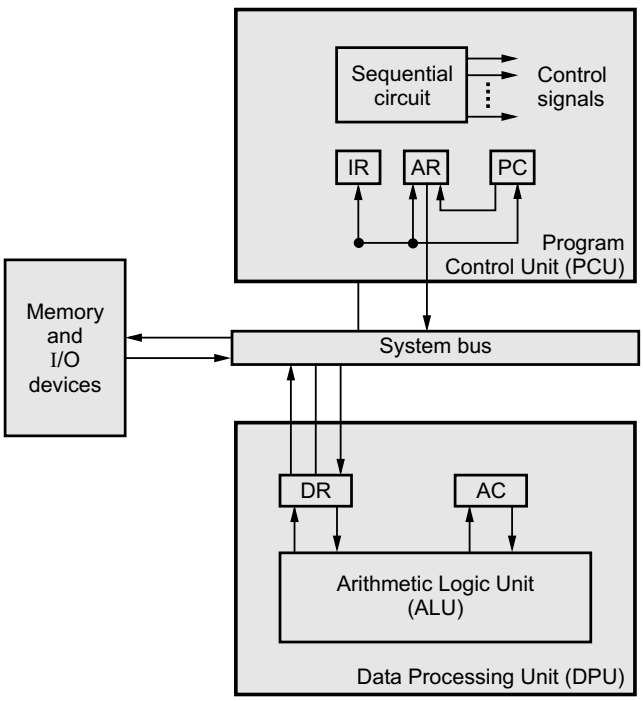


Fig. 3.3.19 Accumulator based CPU organisation

To design the circuit for PCU, it is necessary to first identify the relevant control actions (micro-operations) needed to process the given instruction set using the hardware from Fig. 3.3.19. The Fig. 3.3.20 shows the flowchart which gives the behavioural description of the CPU.

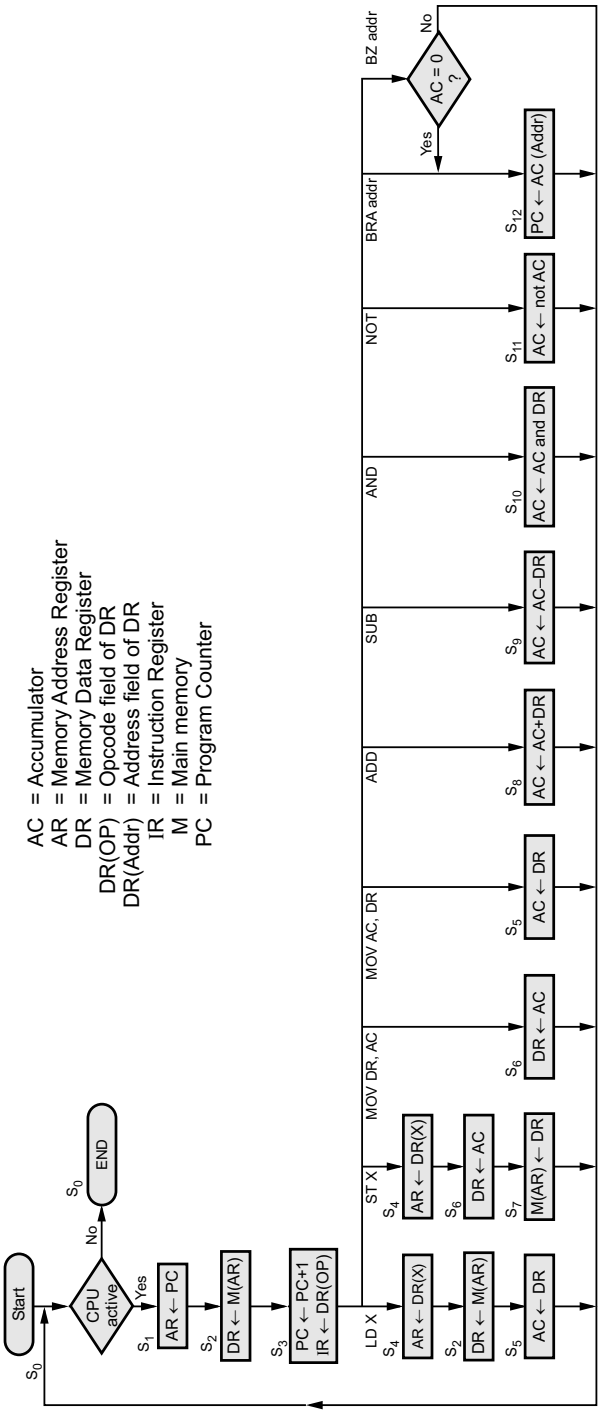


Fig. 3.3.20 Flowchart of the accumulator based CPU

We know that, all instructions require a common instruction-fetch step. It is followed by an execution step that varies with instruction to instruction. In the fetch step, the contents of the PC are copied into the memory Address Register (AR). A memory read operation is then executed, which transfers the instruction word I (opcode) to memory Data Register (DR). This is expressed in the flowchart as DR ← M (AR). The opcode is then transferred to instruction register IR, where it is decoded; at the same time PC is incremented to point to the next consecutive instruction in the memory.

The operations required to execute instruction will depend on the opcode. For example, load instruction LD X is executed in three steps : the address field of LD X is transferred to AR, the contents of the addressed location are transferred to DR by performing memory read operation [DR ← M(AR)], and finally, the contents of DR are transferred to the accumulator AC. As shown in the Fig. 3.3.20, instructions require 1 to 3 steps for execution.

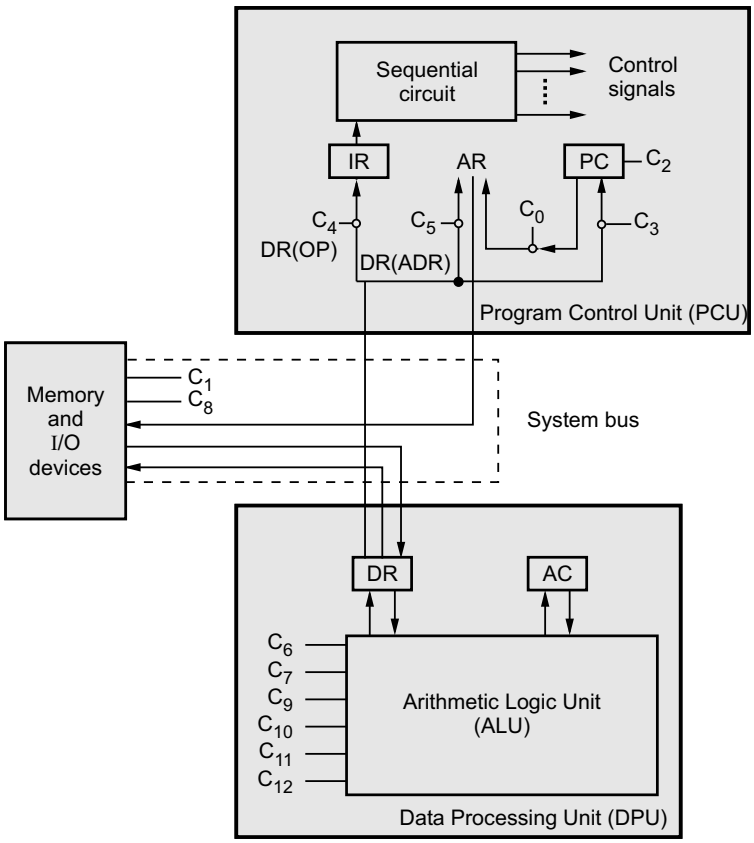


Fig. 3.3.21 Accumulator based CPU with control points

The micro-operations shown in the flowchart implicitly determine the control signals and control points needed by the CPU. The Table 3.3.5 shows the set of control signals for the CPU and their functions, while Fig. 3.3.21 shows the approximate positions of the corresponding control points in both the PCU and DPU.

Control signal	Operation controlled
C_0	AR ← PC
C_1	DR ← M(AR)
C_2	PC ← PC + 1
C_3	PC ← DR (ADR)
C_4	IR ← DR (OP)
C_5	AR ← DR (ADR)
C_6	DR ← AC
C_7	AC ← DR
C_8	M(AR) ← DR
C_9	AC ← AC + DR
C_{10}	AC ← AC – DR
C_{11}	AC ← AC and DR
C_{12}	AC ← not AC

Table 3.3.7 Control signal definitions for the accumulator-based CPU

The control signals shown in Table 3.3.7 can be generated using sequential circuit. This sequential circuit will have 10 input signals from instruction decoder, one per instruction type, one input status signal (AC = 0) and one input start signal. Thus, the sequential circuit will have 12 inputs and 13 outputs. As shown in the Fig. 3.3.21, each distinct action box is assigned to a different state. Therefore, the circuit has 13 states S_0 : S_{12}. Once the inputs, outputs and internal states are known the circuit can be implemented using standard procedure for sequential machine design.

3.3.4 Design of Control Unit of GCD Processor using State Table

Let us see the design of special purpose processor that computes the greatest, common divisor, gcd (A, B) of two positive integers A and B : gcd (AB) is defined as the largest integer that divides exactly into both A and B. For example, gcd (12, 20) = 4, and gcd (13, 21) = 1. We can assume that gcd (0, 0) = 0

To find the gcd of two numbers we have to divide greater number by smaller number, if remainder is zero, divisor is a gcd. If remainder is not zero, remainder becomes new divisor and previous divisor becomes dividend and this process is repeated until we get remainder 0.

For example, if numbers are 20 and 15 we can find gcd as follows :

$20 \div 15 = 1$ Remainder 5 i.e. $\neq 0$

$\therefore \quad 15 \div 5 = 3$ Remainder 0

$\therefore \quad$ gcd = 5

The hardware description language (HDL) algorithm of the above method is as given below. In this program division of two numbers is achieved by repetitive subtraction. In this algorithm the smaller number is subtracted from greater number and result is stored as a new greater number. If greater number is still greater than smaller number subtraction is repeated; otherwise smaller number becomes a new greater number and last remainder becomes a new smaller number. This process is repeated until remainder is zero.

Algorithm

```
gcd (in : A, B ; out : Y)
register RegA, RegB. RegTemp;
      RegA : = A;                          {Read first number}
      RegB  : = B;                         {Read second number }
while RegA > 0 do
      begin
            if     RegA ≤ RegB then        {swap two numbers}
            begin
                  RegTemp : = RegB;
                  RegB : = RegA;
                  RegA : = RegTemp;
            end
       RegA : = RegA – RegB ;   { Subtract smaller number from greater number}
 end
       Y : = RegB ;       {Write the result}
 end gcd;
```

The above algorithm goes through following steps when A = 12 and B = 20.

RegA : = 12, RegB : = 20

Loop 1 : RegA > 0 ✓ RegA ≤ RegB ✓ RegA := 20; RegB = 12 RegA : = RegA – RegB = 8;

Loop 2 : RegA > 0 ✓ RegA ≤ RegB ✓ RegA : = 12; RegB : = 8 RegA : = RegA – RegB = 4;

Loop 3 : RegA > 0 ✓ RegA ≤ RegB ✓ RegA : = 8; RegB : = 4 RegA : = RegA – RegB = 4;

Loop 4 : RegA > 0 ✓ RegA ≤ RegB ✓ RegA : = 4; RegB : = 4 RegA : = RegA – RegB = 0;

Loop 5 : RegA > 0 X

Y := RegB = 4;

Note : ✓ mark indicates True condition

X mark indicates False condition

Therefore, gcd (12, 20) = 4

To implement the gcd processor we need pair of register RegA and RegB to store the corresponding variables, the subtractor to perform subtraction, two magnitude comparator : one to compare two numbers and other to compare greater number with zero, and multiplexer for data routing. This is illustrated in Fig. 3.3.22.

As shown in the Fig. 3.3.22, control unit generates control signals Load A and Load B to load data in Register A and Register B independently. The multiplexer with select inputs (select AB, swap and subtract) decides the inputs for register A and B. With this hardware it is possible to read from and write to a register in the same clock cycle. This allow us to swap the contents of register A and register B without the need of temporary register.

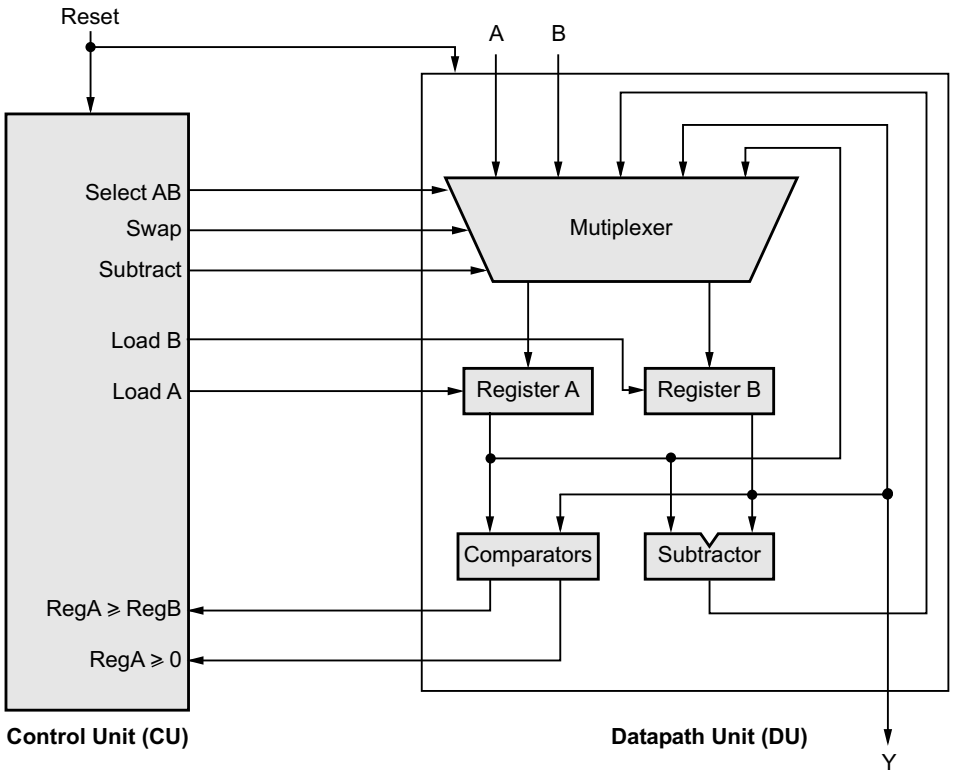


Fig. 3.3.22 Hardware for gcd processor

The control unit provides select AB signal to route A and B, to register A and B, respectively. The swap signal from the control unit controls the swap operation, i.e. A : = B, B : = A. Finally, the subtract signal from the control unit controls the subtraction RegA : = RegA – RegB by routing the output of the subtractor to RegA through multiplexer. The input signals to the control unit are an asynchronous Reset signal, two comparison signals (RegA ≤ RegB) and (RegA > 0) generated by datapath unit and internal clock signal.

We can easily identify the set of states for control unit by examining the behaviour defined in algorithm. These states are indicated in the state diagram shown in Fig. 3.3.23.

A state S_0 is entered when Reset becomes 1; this state also loads A and B into the register A and B, respectively. The subsequent actions of gcd processor are either a swap or a subtraction, for which we assign the states S_1 and S_2, respectively. A final state S_3 indicates that the gcd (A, B) has been computed.

From the state diagram shown in Fig. 3.3.23, we can determine the state table for gcd processor as shown in Table 3.3.8. In state S_0, if input RegA > 0 is not true (RegA > 0 = 0) transition $S_0 \rightarrow S_3$ is made irrespective of the other input Reg A ≤ RegB. On the other hand if input RegA > 0 is true (RegA > 0 = 1) the while loop is entered, and a transition is made to S_1 to perform a swap if RegA ≤ RegB is true (RegA ≤ RegB = 1); otherwise the transition is made to S_2 to perform a subtraction. Since a subtraction always follows swap, all next state entries in the present state S_1 are S_2. The next state entries for state S_2 are same as state S_0. The outputs are activated as shown in the state diagram. The state S_3 is dead state; it is not affected by any input accept reset.

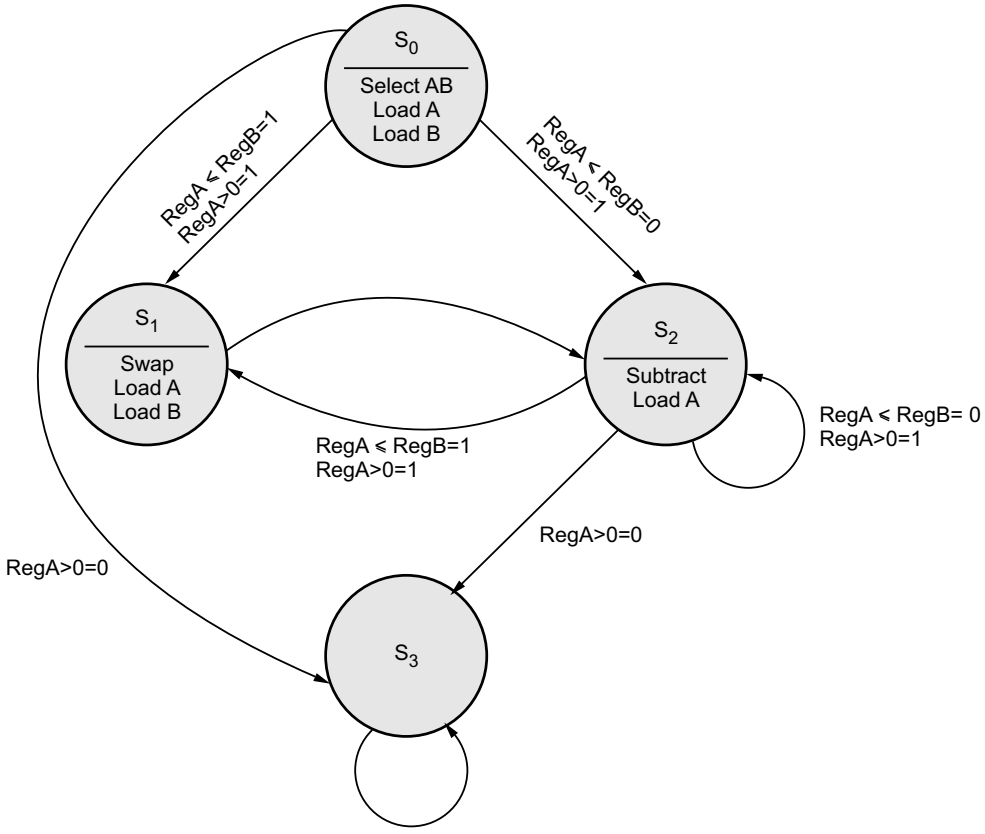


Fig. 3.3.23 State diagram for gcd processor

Present state	Next states with inputs (RegA ≤ RegB, RegA > 0)			Outputs				
	X0	01	11	Subtract	Swap	Select AB	Load A	Load B
S_0	S_3	S_2	S_1	0	0	1	1	1
S_1	S_2	S_2	S_2	0	1	0	1	1
S_2	S_3	S_2	S_1	1	0	0	1	0
S_3	S_3	S_3	S_3	0	0	0	0	0

Table 3.3.8 State for gcd processor

In classical or state table design method next steps are as follows :

1. Assign binary codes to the states.
2. Determine number of flip-flops required.
3. Determine the excitation table.
4. Determine the minimized Boolean function for each flip-flop input.
5. Draw the complete logic diagram.

According to step 1 we assign binary codes to states as follows :

$S_0 = 00$, $S_1 = 01$, $S_2 = 10$ and $S_3 = 11$

As number of states are four we require two flip-flips. Once the binary codes and number of flip-flops are determined we can determine the excitation table as shown in Table 3.3.9. Here, we have used D flip-flops to implement the control unit for the gcd processor.

Present State			Inputs		Next State		Outputs				
	D_1	D_0	RegA ≤ RegB	RegA > 0	D_1^+	D_0^+	Subtract	Swap	Select AB	Load A	Load B
S_0	0	0	X	0	1	1	0	0	1	1	1
	0	0	0	1	1	0	0	0	1	1	1
	0	0	1	1	0	1	0	0	1	1	1
S_1	0	1	X	0	1	0	0	1	0	1	1
	0	1	0	1	1	0	0	1	0	1	1
	0	1	1	1	1	0	0	1	0	1	1
S_2	1	0	X	0	1	1	1	0	0	1	0
	1	0	0	1	1	0	1	0	0	1	0
	1	0	1	1	0	1	1	0	0	1	0
S_3	1	1	X	0	1	1	0	0	0	0	0
	1	1	0	1	1	1	0	0	0	0	0
	1	1	1	1	1	1	0	0	0	0	0

Table 3.3.9 Excitation table for gcd processor

K-map Simplification

For D_0^+

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	1	0	1	1
01	0	0	0	0
11	1	1	1	1
10	1	0	1	1

$D_0^+ = \overline{D}_0\ \overline{RegA>0} + D_1D_0 + \overline{D}_0\ RegA \leqslant RegB$

For D_1^+

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	1	1	0	1
01	1	1	1	1
11	1	1	1	1
10	1	1	0	1

$D_1^+ = \overline{RegA \leqslant RegB} + D_0 + \overline{RegA > 0}$

For subtract

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	1	1	1

Subtract = $D_1\ \overline{D}_0$

For swap

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

Swap = $\overline{D}_1\ D_0$

For select AB

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Select AB = $\overline{D}_1\ \overline{D}_0$

For load A

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

Load A = $\overline{D}_1 + \overline{D}_0$

For Load B

RegA ⩽ RegB, RegA>0

D_1D_0	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

Load B = $\overline{D}_1$

Fig. 3.3.24

Logic Diagram

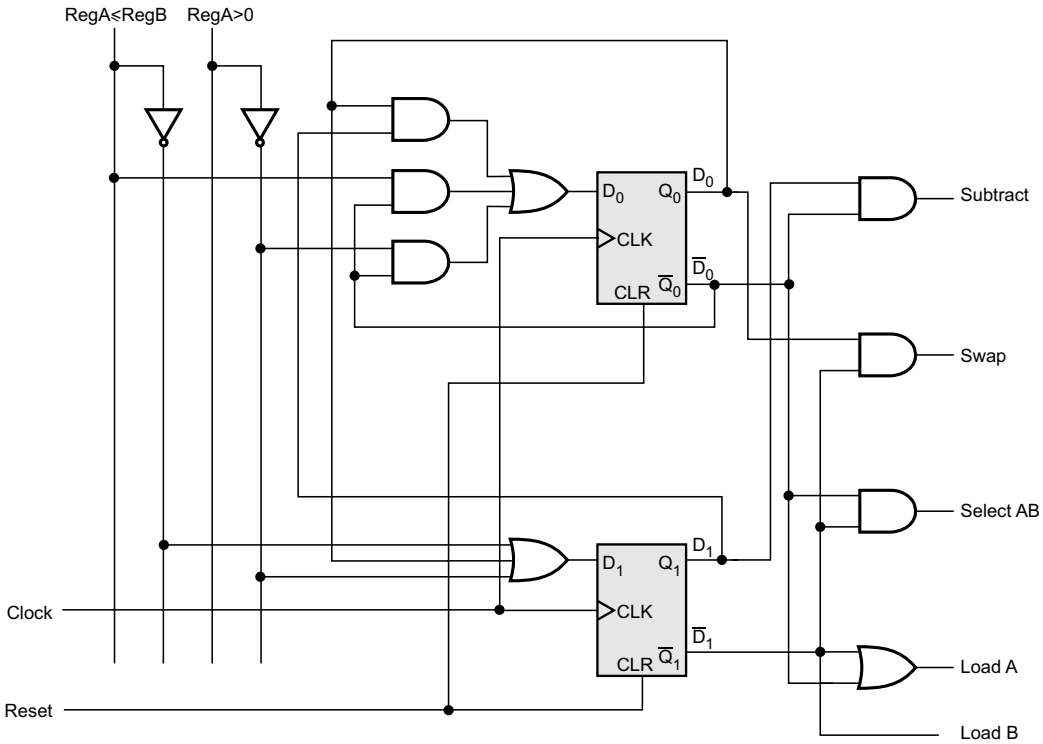


Fig. 3.3.25 Classical design for the control unit of the gcd processor

3.3.5 Design of Control Unit of GCD Processor using One Hot Method

The classical method used for the design of sequential circuit minimizes a control unit's memory elements; however such design is more complicated and due to that design debugging and subsequent maintenance of the circuit become difficult. An alternative approach that simplifies the design process and gives combinational circuit a regular and predictable structure, is the one hot method. In this method, the binary value of each state contains a single 1-the hot bit - while all the remaining bits are 0, hence the name. Thus the state assignment for a four-state machine like the control unit of the gcd processor has the following state assignments.

$S_0 = 0001$, $S_1 = 0010$, $S_2 = 0100$ and $S_3 = 1000$

In general, n flip-flops are required to represent n-states. For the same reason, the one-hot method is restricted to fairly small values of n. The important feature of this technique is that the next-state and output equations have a simple, symbolic form and can be written down directly from the control units state table.

In one hot design we know that n flip-flops are required to represent n-states. If we use D flip-flops then state S_i corresponds to hot variable D_i. If we assume that $I_{j,1}$, $I_{j,2}$, $I_{j,n}$ denote all input combinations that cause a state transition from S_j to S_i. Then each AND combination of the form D_j, I_{jk} must make $D_i = 1$. Hence, considering all such combinations that cause transition to S_i, we can write

$$D_i^+ = D_1 (I_{1,1} + I_{1,2} + \ldots + I_{1,n_1}) + D_2 (I_{2,1} + I_{2,2} + \ldots + I_{1,n_2}) + \quad \ldots (3.3.1)$$

This gives the Boolean expression in the SOP form

$$D_i^+ = D_1 I_{1,1} + D_1 I_{1,2} + \ldots + D_1 I_{1,n_1} + D_2 I_{2,1} + D_2 I_{2,2} + \ldots + D_2 I_{2,n_2} + \ldots$$

which can be implemented using an AND-OR or NAND-NAND logic. Let us see the implementation of the control unit of the gcd processor using one-hot method.

Refer the state diagram for the control unit of the gcd processor given in Fig. 3.3.23. State S_1 appears as a next state only for S_0 and S_2 in each case input combination $(\text{RegA} \geq \text{RegB}) \cdot (\text{RegA} > 0)$. Hence equation (1) becomes

$$D_1^+ = D_0 \cdot (\text{RegA} \leq \text{RegB}) \cdot (\text{RegA} > 0) + D_2 \cdot (\text{RegA} \leq \text{RegB}) \cdot (\text{RegA} > 0)$$

Applying similar method we can write the entire set of next-state and output equations for the gcd processor as given below

$$D_1^+ = 0$$

$$D_1^+ = D_0 \cdot (\text{Reg A} \leq \text{Reg B}) \cdot (\text{RegA} > 0) + D_2 \cdot (\text{RegA} \leq \text{RegB}) \cdot (\text{RegA} > 0)$$

$$D_2^+ = D_0 \cdot (\overline{\text{RegA} \leq \text{RegB}}) \cdot (\text{RegA} > 0) + D_1 + D_2 \cdot (\overline{\text{RegA} \leq \text{Reg B}}) \cdot (\text{RegA} > 0)$$

$$D_3^+ = D_0 \cdot (\overline{\text{RegA} > 0}) + D_2 + (\overline{\text{RegA} > 0}) + D_3$$

$$\text{Subtract} = D_2$$

$$\text{Swap} = D_1$$

$$\text{Select AB} = D_0$$

$$\text{Load A} = D_0 + D_1 + D_2$$

$$\text{Load B} = D_0 + D_1$$

The Fig. 3.3.26 shows the logic diagram for the control unit of the gcd processor using one hot design method.

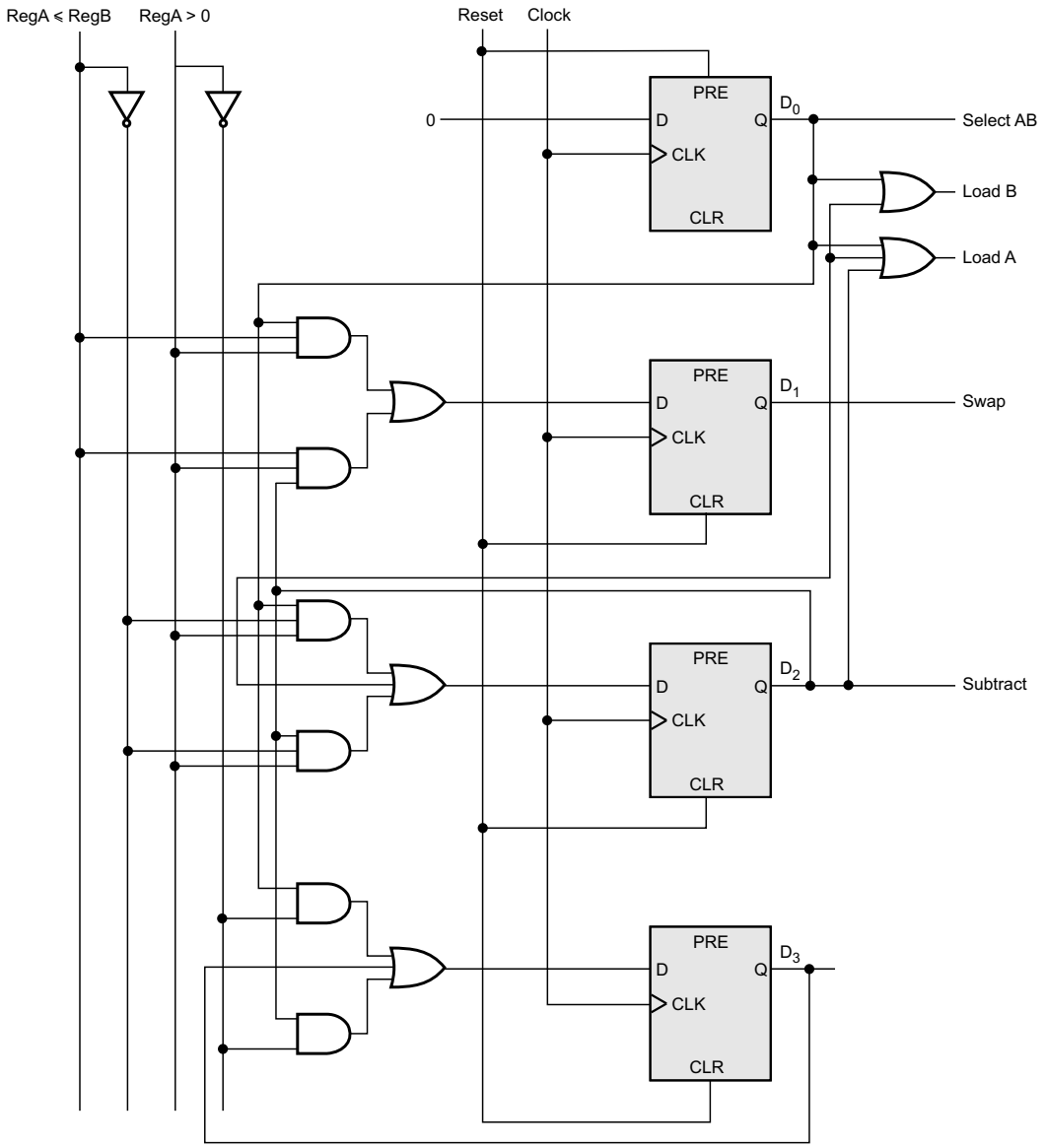


Fig. 3.3.26 One hot design for the control unit of the gcd processor

Review Questions

1. *Describe the control unit organization with a separate encoder and decoder functions in a hardwired control.* **May-11, Marks 8**
2. *Generate the logic circuit for the following functions and explain.*
   $Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + ...$
   $END = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot BRN + ...$ **May-11, Marks 8**
3. *Briefly describe the design of a hardwired control unit.* **May-10, Marks 10**
4. *Give the organization of a typical hardwired control unit and explain the functions performed by the various blocks. Discuss the data flow for a sample instruction.* **May-07, Marks 5**
5. *Explain the operation of hardwired control unit.* **Dec.-08, Marks 16**
6. *State the advantages and disadvantages of hardwired control unit.* **Dec.-07, May-10, Marks 4**
7. *Explain the various design methods of hardwired control unit.* **Dec.-06, Marks 8**
8. *Briefly describe the design of a hardwired control unit.* **May-10, Marks 10**
9. *Explain the design of multiplier control unit using delay element method.*
10. *Design control unit for divider circuit using sequence counter method.*
11. *Explain the PLA method of design of control unit.*
12. *Draw and explain the block diagram of a complete processor.* **May-10, Marks 6**
13. *Describe the basic structure of a pipeline processor and explain how it is carried out in a floating point adder.* **Dec.-03, Marks 16**
14. *Design a micro programmed control unit of non-pipelined general purpose computers.* **Dec.-11, Marks 16**
15. *Draw and explain the hardware required for gcd processor.*
16. *Explain the control unit design for gcd processor using state table method.*
17. *Explain the control unit design for gcd processor using one hot design method.*

3.4 Microprogrammed Control

May-06, 07, 08, 09, 10, 11, 12
Dec.-06, 07, 09, 12

Every instruction in a processor is implemented by a sequence of one or more sets of concurrent microoperations. Each microoperation is associated with a specific set of control lines which, when activated, causes that microoperation to take place. Since the number of instructions and control lines is often in the hundreds, the complexity of hardwired control unit is very high. Thus, it is costly and difficult to design. Further more, the hardwired control unit is relatively inflexible because it is difficult to change the design, if one wishes to correct design error or modify the instruction set.

Microprogramming is a method of control unit design in which the control signal selection and sequencing information is stored in a ROM or RAM called a **control**

memory CM. The control signals to be activated at any time are specified by a **microinstruction**, which is fetched from CM in much similar way an instruction is fetched from main memory. Each microinstruction also explicitly or implicitly specifies the next microinstruction to be used, thereby providing the necessary information for sequencing. A sequence of one or more microoperations designed to control specific operation, such as addition, multiplication is called a **microprogram**. The microprograms for all instructions are stored in the control memory.

The address where these microinstructions are stored in CM is generated by **microprogram sequencer/microprogram** controller. The microprogram sequencer generates the address for microinstruction according to the instruction stored in the IR.

Fig. 3.4.1 shows the microprogrammed control unit. It consists of control memory, control address register, micro instruction register and microprogram sequencer.

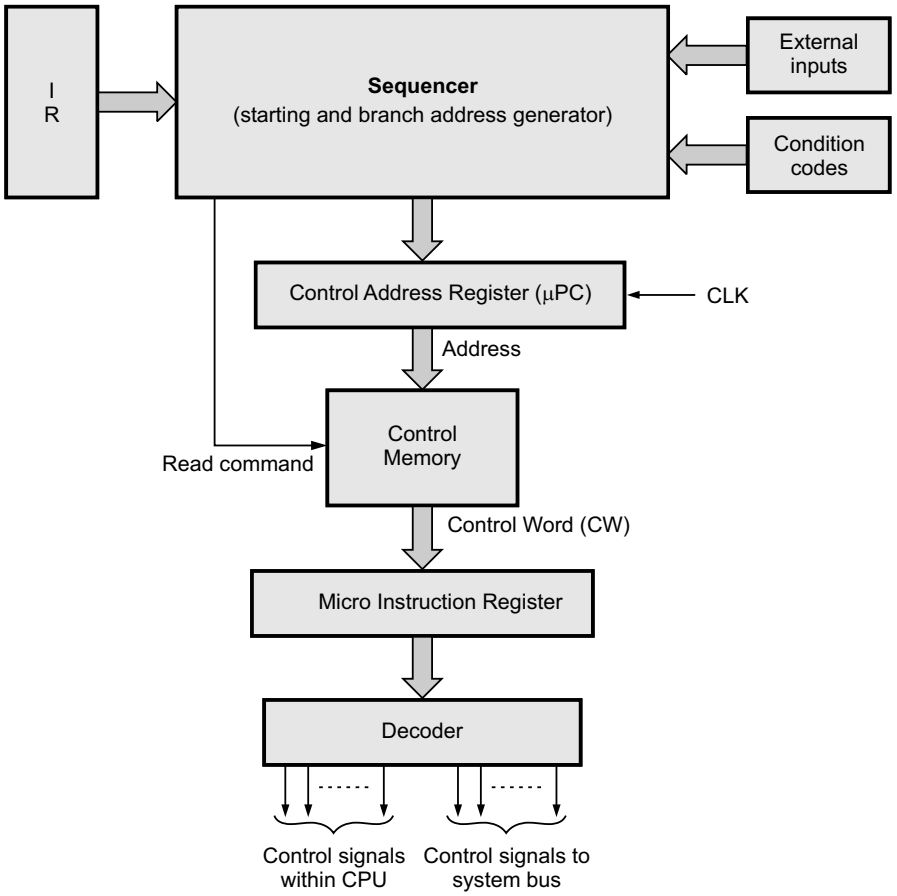


Fig. 3.4.1 Microprogrammed control unit

The components of control unit work together as follows :

- The control address register (μpc) holds the address of the next microinstruction to be read. Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the μpc.
- When address is available in control address register, the sequencer issues READ command to the control memory.
- After issue of READ command, the word from the addressed location is read into the microinstruction register.
- The μpc is then automatically incremented by the clock, causing successive microinstructions to be read from the control memory.
- The content of the micro instruction register generates control signals which are delivered to various parts of the processor in the correct sequence.

Number of times the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action. In such situation, microprogrammed control use conditional branch microinstructions. In additions to the branch address, these instructions specify which of the external inputs, condition codes, or, possibly bits of the instruction register should be checked as a condition for branching to take place.

Let us see the implementation of instruction Branch < 0, as shown in Fig. 3.4.2. When this instruction is loaded into IR, a branch microinstruction transfers control to the corresponding microroutine, which is assumed to start at location 45 in the control memory. This address is the output of the starting address generator block in Fig. 3.4.1. The microinstruction at location 45 tests the N bit of the condition codes. If this bit is equal to 0, a branch takes place to location 0 to fetch a new machine instruction. Otherwise, the microinstruction at location 46 is executed to put the branch target address into register Z. The microinstruction in location 47 loads this address into the PC.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
1	Z_{out}, PC_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
45	If N=0, then branch to microinstruction 0
46	offset_field_of_IR_{out}, SelectY, Add, Z_{in}
47	Z_{out}, PC_{in}, End

Fig. 3.4.2 Microroutine for the instruction branch < 0

To support microprogram branching, the starting address and branch address generator block loads a new address into the μpc when a microinstruction instructs it to do so. To allow implementation of conditional branch, the external inputs, condition codes and the contents of IR are given to the starting address and branch address generator block. Let us see how the μpc is used at different situations.

1. When a new instruction is loaded into the IR, the μpc is loaded with the starting address of the microroutine for that instruction.
2. When a Branch microinstruction is encountered and the branch condition is satisfied, the μpc is loaded with the branch address.
3. When the End instruction is encountered, the μpc is loaded with the address of the first control word (CW) in the microroutine for the instruction fetch cycle, i.e. address 0.
4. In any other situation, the μpc is incremented every time a new microinstruction is fetched from the control memory.

Advantages of Microprogrammed Control

- It simplifies the design of control unit. Thus it is both, cheaper and less error prone to implement.
- Control functions are implemented in software rather than hardware.
- The design process is orderly and systematic.
- More flexible, can be changed to accommodate new system specifications or to correct the design errors quickly and cheaply.
- Complex function such as floating point arithmetic can be realised efficiently.
- The new or modified instruction set of CPU can be easily implemented by simply rewriting or modifying the contents of control memory.
- The fault can be easily diagnosed in the micro-program control unit using diagnostics tools by maintaining the contents of flags, registers and counters.

Disadvantages of Microprogrammed Control

- A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.
- The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.
- The design duration of micro-program control unit is more than hardwired control unit for smaller CPU.

Besides these disadvantages, the microprogramming is the dominant technique for implementing control units.

Example 3.4.1 *Write a combined micro routine that can implement that BGT (Branch if > 0), BPL (Branch if plus), and BR (Branch Unconditionally) instructions. The branch conditions, for the BGT and BPL instructions are Z + (N XOR V) = 0 and N = 0, respectively. What is the total number of micro instructions required ? How many micro instructions are needed if a separate micro routine is used for each machine instruction ?*

Solution : For bus organisation, we write microroutine for the implementation of BGT (Branch if > 0), BPL (Branch if plus) and BR (Branch unconditionally) instructions as follows.

Combine Microroutine

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Y_{in}, selectC, Add, Z_{in}
1	Z_{out}, PC_{in}, WMFC
2	MDR_{out}, IR_{in}

According to opcode in IR branch to starting address of appropriate microroutine.

Unconditional Branch

15	Offset_field_of_IR_{out}, selectY, Add, Z_{in}
16	Z_{out}, PC_{in}, End

Branch if plus

25	If N = 1 then branch to microinstruction 0
26	Offset_field_of_IR_{out}, selectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Branch if > 0

35	If Z + (N XOR V) = 1 then branch to microinstruction 0
16	Offset_field_of_IR_{out}, selectY, Add, Z_{in}
2	Z_{out}, PC_{in}, End

The first three microinstructions in the combine microroutine are used to fetch the opcode. The total number of microinstructions required are 11. If a separate microroutine is used for each machine instruction 17 microinstructions are needed.

3.4.1 Microinstruction

A simple way to structure microinstructions is to assign one bit position to each control signal required in the CPU. However, this scheme has one serious drawback-assigning individual bits to each control signal results in long microinstructions, because the number of required signals is usually large. Moreover, only a few bits are used in any given instruction. The solution of this problem is to group the control signals.

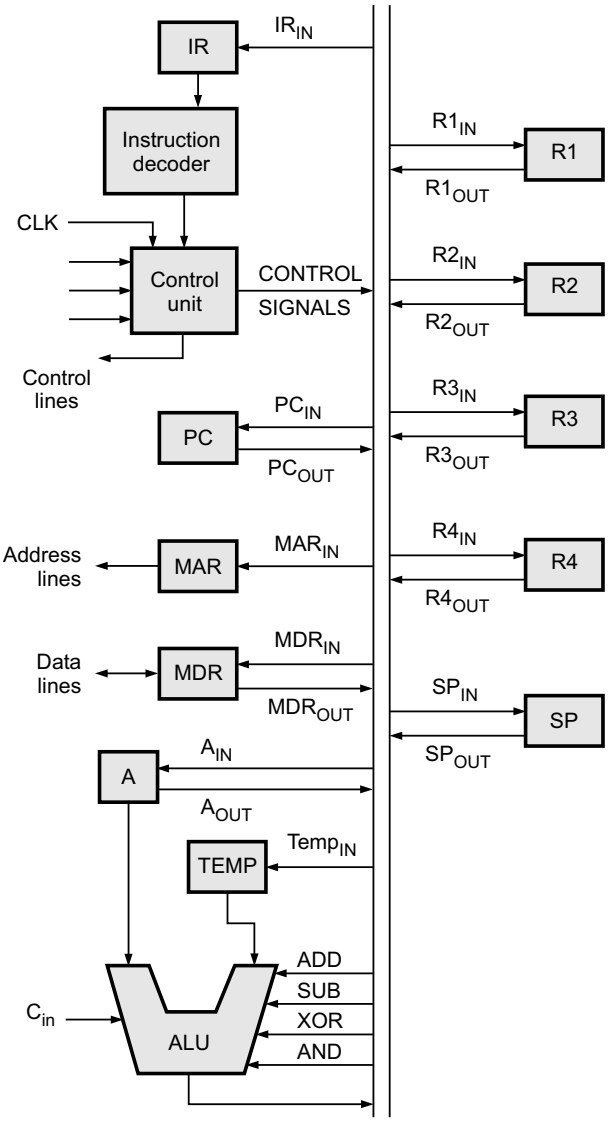


Fig. 3.4.3 Single bus CPU structure with control signals

3.4.1.1 Grouping of Control Signals

Grouping technique is used to reduce the number of bits in the microinstruction. This technique is explained in the following section.

Let us consider single bus CPU having different control signals, as shown in Fig. 3.4.3. (Fig. 3.4.3 see on previous page)

Gating signals : (IN and OUT signals).

Control signals : Read, write, clear A, set carry in, continue operation, WMFC, end, etc.

ALU Signals : Add, sub, etc. There are in all 39 signals and hence each microinstruction will have 39 bits. It is not at all necessary to use all 39 bits for every microinstruction because by using grouping of control signals we minimize number of bits for microinstruction.

Ways to reduce number of bits in microinstruction

1. Most signals are not needed simultaneously.
2. Many signals are mutually exclusive e.g. Only one function of ALU can be activated at a time.
3. A source for data transfers must be unique which means that it should not be possible to get the contents of two different registers on to the bus at the same time.
4. Read and write signals to the memory can't be activated simultaneously.

This suggests the possibility of grouping the control signals so that all the signals that are mutually exclusive are placed in the same group. Thus a group can specify one microinstruction at a time. So with this suggestions 39 control signals can be grouped in 8 different groups.

G_1 (4 Bits) : IN grouping		G_2 (4 Bits) : OUT grouping	
0000	No Transfer	0000	No Transfer
0001	IR_{IN}	0001	PC_{OUT}
0010	PC_{IN}	0010	MDR_{OUT}
0011	MDR_{IN}	0011	R_{1OUT}
0100	MAR_{IN}	0100	R_{2OUT}
0101	A_{IN}	0101	R_{3OUT}
0110	$Temp_{IN}$	0110	R_{4OUT}
0111	R_{1IN}	0111	SP_{OUT}
1001	R_{3IN}		

1010	R_{4IN}		
1011	SP_{IN}		

G_3 (4 Bits) : ALU Functions	G_4 (2 Bits) : RD/WR Control Signals
0000 ADD 0001 SUB ⋮ 16 ALU Functions ⋮ 1111 XOR	00 No Action 01 Read 10 Write

G_5 (1 Bit) : A Register	G_6 (1 Bit) : Carry
0 - No action	0 - Carry in to ALU = 0
1 - Clear A	1 - Carry in to ALU = 1
G_7 (1 bit) : Operation	**G_8 (1 bit) : Operation**
0 : No action 1 : WMFC	0 : Continue Operation 1 : End

The total number of grouping bits are 18. Therefore, we minimized 39 bits microinstruction to 18 bit microinstruction. Grouping of control signals result in relatively small increase in the required hardware as it becomes necessary to use decoding circuits to translate the bit patterns of each group into actual control signals. Since the number of bits required is less for microinstruction, less space is required for microinstructions for the instruction.

3.4.1.2 Techniques of Grouping of Control Signals

The grouping of control signals can be done either by using technique called **vertical organisation** or by using technique called **horizontal organisation**. Highly encoded scheme that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a **vertical organisation**. On the other hand, the minimally encoded scheme, in which resources can be controlled with a single instruction, is called a **horizontal organisation**.

The Table 3.4.1 shows the comparison between horizontal and vertical organisation.

Sr. No.	Horizontal	Vertical
1.	Long formats.	Short formats.
2.	Ability to express a high degree of parallelism.	Limited ability to express parallel microoperations.

3.	Little encoding of the control information.	Considerable encoding of the control information.
4.	Useful when higher operating speed is desired.	Slower operating speeds.

Table 3.4.1

The advantages and disadvantages of horizontal and vertical organisations can be summarized as follows :

1. The horizontal organisation approach is suitable when operating speed of computer is a critical factor and where the machine structure allows parallel usage of a number of resources.
2. Vertical approach results in slower operations speed but less bits are required in the microinstruction.
3. In vertical approach the significant factor is the reduced requirement for the parallel hardware required to handle the execution of microinstructions.

3.4.2 Microinstruction Sequencing

The task of microprogram sequencing is done by microprogram sequencer. There are two important factors that must be considered while designing the microprogram sequencer :

- The size of the microinstruction and
- The address generation time

The size of the microinstruction should be minimum so that the size of control memory required to store microinstructions is also less. This reduces the cost of control memory. With less address generation time, microinstructions can be executed in less time, resulting better throughput.

During execution of a microprogram, the address of the next microinstruction to be executed has three sources :

- Determined by instruction register
- Next sequential address
- Branch.

Out of these three address sources, first occurs only once per instruction cycle. The second source is most commonly used. However, if we store separate microinstructions for each machine instruction, there will be large number of microinstructions. As a result, large space in CM is required to store these microinstructions. We know that, machine instructions involve several addressing modes and there can be many instructions and addressing mode combinations. A separate microinstructions for each of these combinations would produce considerable duplication of common microinstructions. We want to organize the microprograms such that they share as many

microinstructions as possible. This requires many branch microinstructions, both unconditional and conditional to transfer control amongst the various microinstructions. Thus it is important to design compact, time-efficient techniques for microinstruction branching.

Let us see how microinstructions can be shared using microinstruction branching. Consider instruction ADD R_{src}, R_{dst}. The instruction adds the source operand to the contents of register R_{dst} and places the sum in R_{dst}, the destination register. Let us assume that the source operand can be specified in the following addressing modes : Indexed, autoincrement, autodecrement, register indirect and register direct. We now use this instruction in conjunction with the CPU structure shown in Fig. 3.4.4 to demonstrate a possible microprogrammed implementation. Fig. 3.4.5 shows a flowchart of a microprogram for the ADD R_{src}, R_{dst} instruction. Each box in the flowchart corresponds to a microinstruction that controls the transfers and operations indicated within the box. The microinstruction is located at the address indicated by the number above the upper right-hand corner of the box. During the execution of the microinstruction, the branching takes place at point A. The branching address is determined by the addressing mode used in the instruction.

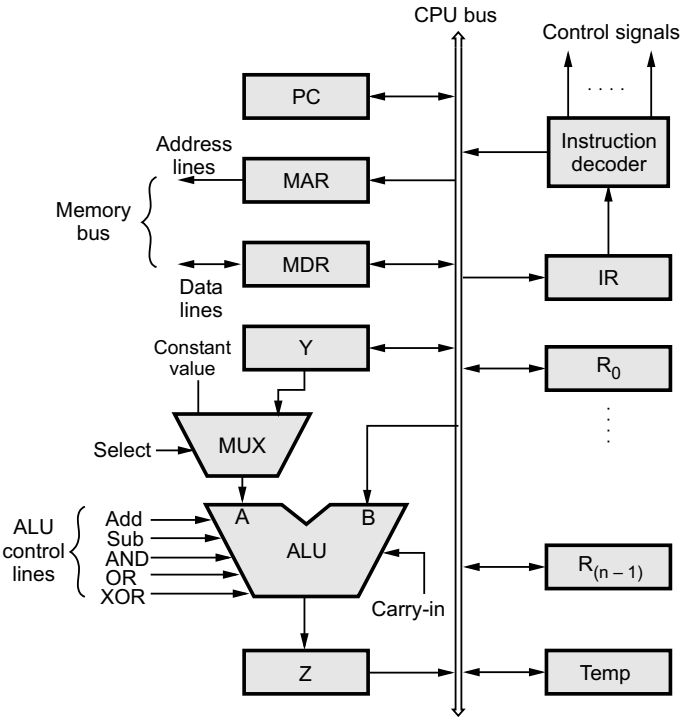


Fig. 3.4.4 Single bus organisation of processor

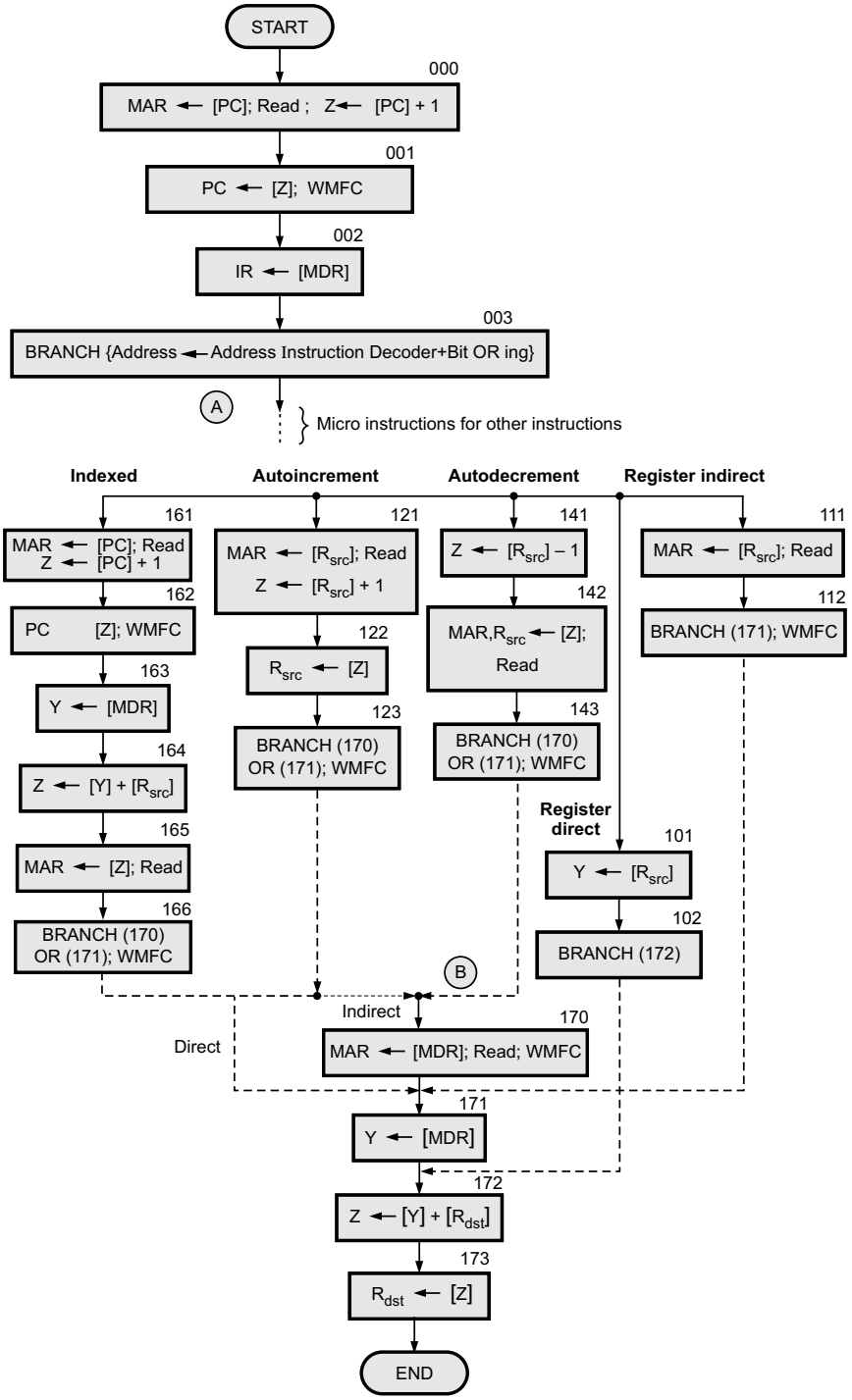


Fig. 3.4.5 Flowchart of a microprogram for the ADD (R_{src})+, R_{dst} instruction

At point B, it is necessary to choose between actions required by direct and indirect addressing modes. If the indirect mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory. If the direct mode is specified, this fetch must be bypassed by branching immediately to location 171. The remaining microoperations required to complete execution of instruction are same and hence shared by the instructions having operand with different addressing modes.

From the above discussion we can say that branching allows sharing of microinstructions for different microprograms and it reduces the size of control memory.

3.4.3 Techniques for Modification or Generation of Branch Addresses

Bit-ORing

In this technique, the branch address is determined by ORing particular bit or bits with the current address of the microinstruction. For example, if the current address is 170 and the branch address is 172 then the branch address can be generated by ORing 02 (bit 1), with the current address.

Using Condition Variables

In this technique the condition variables are used to modify the contents of the CM address register directly, thus eliminating whole or in part the need for branch addresses in microinstructions. For example, let the condition variable CY indicate occurrence of CY = 1 and no carry when CY = 0. Suppose that we want to execute a SKIP_ON_CARRY microinstruction. This can be done by logically connecting CY to the count enable input of μpc at on appropriate point in the microinstruction cycle. This allows the overflow condition to increment μpc an extra time, thus performing the desired skip operation.

Wide-Branch Addressing

Generating branch addresses becomes more difficult as the number of branches increases. In such situations Programmable Logic Array can be used to generate the required branch addresses. This simple and inexpensive way of generating branch addresses is known as wide-branch addressing. Here, the opcode of a machine instruction is translated into the starting address of the corresponding micro-routine. This is achieved by connecting the opcode bits of the instruction register as inputs to the PLA, which acts as a decoder. The output of the PLA is the address of the desired microroutine.

Examples with Solutions

Example 3.4.2 *For a single bus organisation of CPU, write a microprogram for instruction. Add (R_{src})+, R_{dst}.*

Solution : Let us examine the path needed for the flowchart in 3.4.5 to execute the instruction Add(R_{src})+, R_{dst}.

In this instruction the source operand is accessed in the autoincrement mode and the R_{src} and R_{dst} are general purpose registers in the processor. We assume that the processor has 16 registers that can be used for addressing purpose, each specified using a 4-bit code. We also assume that the instruction has a 3-bit field, (bits 8-10) used to specify the addressing mode for the source operand, as shown in the Fig. 3.4.6. Bit patterns 11,10,01 and 00 located in bits 10 and 9 denote the indexed, autodecrement, autoincrement and register modes respectively. For each of these modes bit-8 is used to specify the indirect version. For example, 100 in the mode field specifies the direct version of the autodecrement mode, whereas 101 specifies the indirect version.

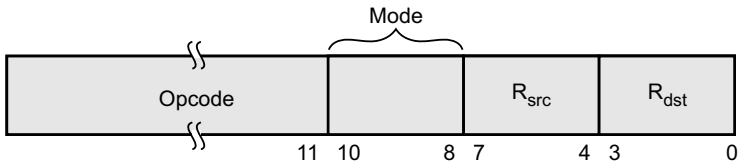


Fig. 3.4.6 Contents of IR

As a part of execution, first the opcode and mode fields are decoded to determine that an R_{src} or R_{dst} register is involved. The decoded output is then used to gate the contents of the R_{src} or R_{dst} fields in the IR into a second decoder, which produces the gating signals for the actual registers R_0 to R_{15}.

The flowchart of a microprogram for the ADD R_{src}, R_{dst} instruction in Fig. 3.4.5 is drawn by combining the microroutines for all possible values of the mode field, resulting in a structure that requires many branch points. The instruction Add (R_{src})+, R_{dst} requires two branch microinstructions. In each branch microinstruction, the expression in brackets indicates the branch address that is to be loaded into the μpc and how this address is modified using the bit-ORing scheme. For example, the branch instruction at location 123 modifies the branch address 170 to 171 by ORing the bit-8 in the IR with bit μpc0 to change the addressing mode from indirect to direct, as shown in Fig. 3.4.7.

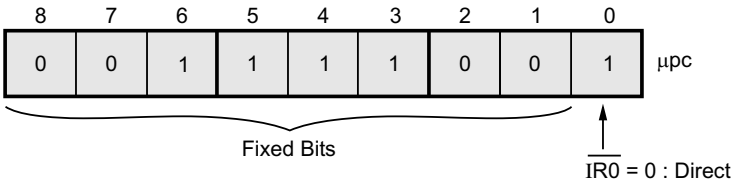


Fig. 3.4.7

The address for branch microinstruction at location 003 is generated as shown in Fig. 3.4.8.

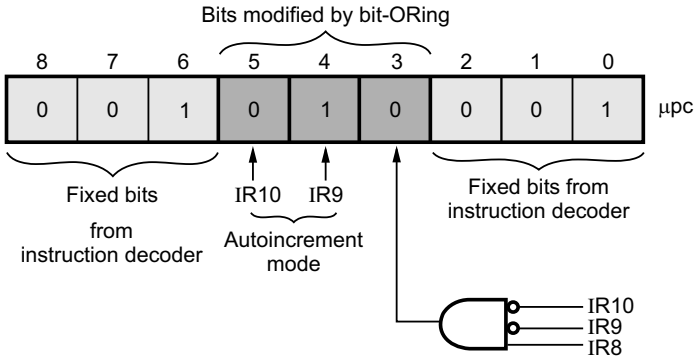


Fig. 3.4.8 Bit ORing for branch microinstruction at location 003

The Table 3.4.2 gives the microinstruction sequence for the execution of Add $(R_{src})+$, R_{dst} instruction.

Address (Octal)	Microinstruction
0 0 0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
0 0 1	Z_{out}, PC_{in}, MWFC
0 0 2	MDR_{out}, IR_{in}
0 0 3	μBranch $\left\{\begin{array}{l}\mu pc \leftarrow (101)_8 \text{ from instruction decoder} \\ \mu pc_{5,4} \leftarrow \left[IR_{10,9}\right],\ \mu pc_3 \leftarrow \left[\overline{IR}_{10}\right] \cdot \left[\overline{IR}_{9}\right] \cdot \left[IR_{8}\right]\end{array}\right\}$ i.e. $\mu pc_{5,4,3} \leftarrow (010)_2$
⋮	
1 2 1	R_{srcout}, MAR_{in}, Read, SelectC, Add, Z_{in}
1 2 2	Z_{out}, R_{srcin}
1 2 3	μBranch $\{\mu pc \leftarrow (170)_8$ from instruction decoder $\mu pc_0 \leftarrow \overline{IR}_8\}$, WMFC
⋮	
1 7 1	MDR_{out}, Y_{in}
1 7 2	R_{dstout}, SelectY, Add, Z_{in}
1 7 3	Z_{out}, R_{dstout}, End

Table 3.4.2 Microinstruction for Add $(R_{src})+$, R_{dst}

Example 3.4.3 *For a single bus organisation of CPU draw flowchart of a microprogram for instruction Add (R_{src})+, R_{dst}.*

Solution : Flowchart

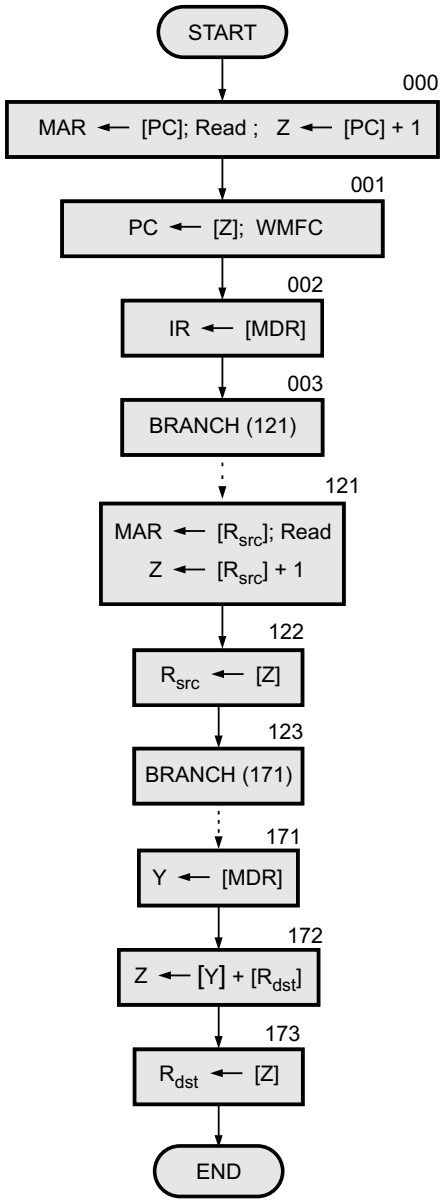


Fig. 3.4.9

Example 3.4.4 *For a single bus organisation of data paths inside the CPU, write a microprogram of micro-instructions and draw chart of a microprogram for the following instruction. MOV* $(R_{src})+$, R_{dst}.

R_{dst} *uses direct and* R_{src} *autoincrement addressing.*

Solution : Flowchart

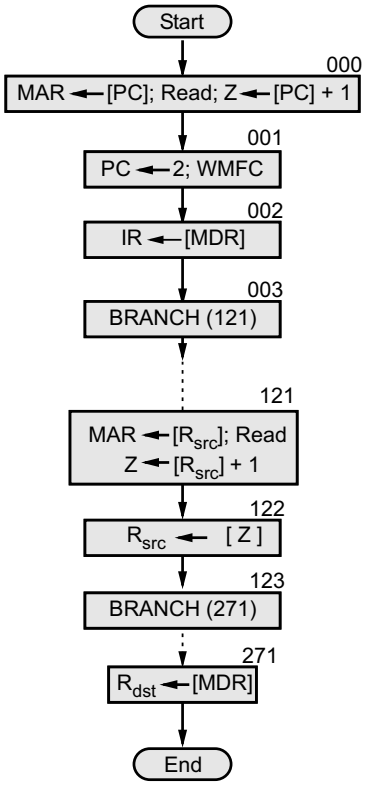


Fig. 3.4.10

Address (Octal)	Microinstruction
0 0 0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
0 0 1	Z_{out}, PC_{in}, MWFC
0 0 2	MDR_{out}, IR_{in}
0 0 3	μBranch $\left\{ \mu pc \leftarrow (101)_8 \text{ from instruction decoder}, \; \mu pc_{5,4} \leftarrow [IR_{10,9}], \; \mu pc_3 \leftarrow [\overline{IR}_{10}] \cdot [\overline{IR}_9] \cdot [IR_8] \right\}$
⋮	

1 2 1	R_{srcout}, MAR_{in}, Read, SelectC, Add, Z_{in}
1 2 2	Z_{out}, R_{srcin}
1 2 3	μBranch{μpc $\leftarrow (270)_8$ from instruction decoder $\mu pc_0 \leftarrow \overline{IR}_8$}, WMFC
⋮	
2 7 1	MDR_{out}, R_{dstin}, End

Example 3.4.5 *For a single bus organisation of CPU draw a flowchart and write microinstructions for instruction ADD (R_{src}), R_{dst}.*

Solution : Flowchart

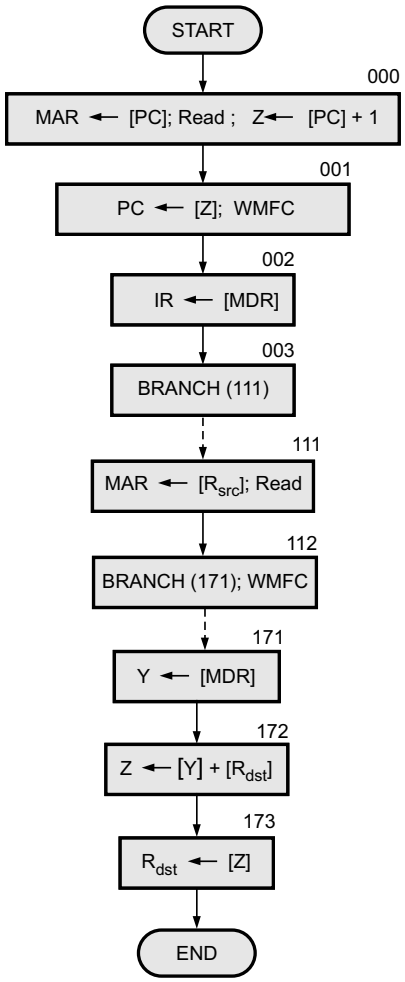


Fig. 3.4.11

Microinstructions

Address (Octal)	Microinstruction
0 0 0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
0 0 1	Z_{out}, PC_{in}, MWFC
0 0 2	MDR_{out}, IR_{in}
0 0 3	$\mu Branch \begin{cases} \mu pc \leftarrow (101)_8 \text{ from instruction decoder} \\ \mu pc_{5,4} \leftarrow [IR_{10,9}],\ \mu pc_3 \leftarrow [\overline{IR}_{10}] \cdot [\overline{IR}_9] \cdot [IR_8] \end{cases}$ i.e. $\mu pc_{5,4,3} \leftarrow (001)_2$
1 1 1	R_{srcout}, MAR_{in}, Read
	$\mu Branch \{\mu pc \leftarrow (170)_8$ from instruction decoder $\mu pc_0 \leftarrow \overline{IR}_8\}$, WMFC
1 1 2	$\mu Branch \{\mu pc \leftarrow (170)_8$ from instruction decoder $\mu pc_0 \leftarrow \overline{IR}_8\}$, WMFC
⋮	
1 7 1	MDR_{out}, Y_{in}
1 7 2	R_{dstout}, SelectY, Add, Z_{in}
1 7 3	Z_{out}, R_{dstin}, End

Table 3.4.3 Microinstruction for Add (R_{src}), R_{dst}

Example 3.4.6 *For a single bus organisation of CPU draw a flowchart and write microinstructions for instruction ADD–(R_{src}), R_{dst}.*

Solution : Flowchart : See on next page.

Address (Octal)	Microinstruction
0 0 0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
0 0 1	Z_{out}, PC_{in}, MWFC
0 0 2	MDR_{out}, IR_{in}
0 0 3	$\mu Branch \begin{cases} \mu pc \leftarrow (101)_8 \text{ from instruction decoder} \\ \mu pc_{5,4} \leftarrow [IR_{10,9}],\ \mu pc_3 \leftarrow [\overline{IR}_{10}] \cdot [\overline{IR}_9] \cdot [IR_8] \end{cases}$ i.e. $\mu pc_{5,4,3} \leftarrow (100)_2$
1 4 1	R_{srcout}, SelectC, Sub, Z_{in}

1 4 2	MAR_{in}, R_{srcin}, Z_{out}, Read
1 4 3	μBranch {μpc ← $(170)_8$ from instruction decoder $μpc_0$ ← $\overline{IR}_8$}, WMFC
⋮	
1 7 1	MDR_{out}, Y_{in}
1 7 2	R_{dstout}, SelectY, Add, Z_{in}
1 7 3	Z_{out}, R_{dstin}, End

Table 3.4.4 Microinstruction for Add –(R_{src}), R_{dst}

Flowchart

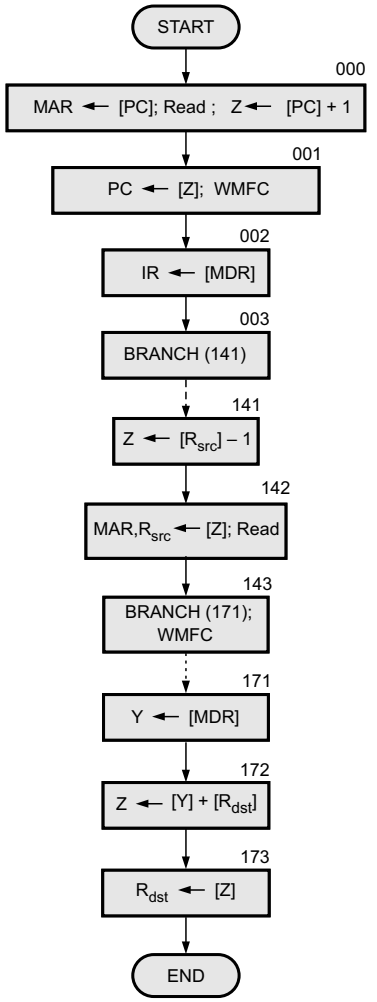


Fig. 3.4.12

Example 3.4.7 *For a single bus organisation to CPU draw a flowchart and write microinstruction for instruction.*

ADD const (R_{src}), R_{dst} where const. is a constant value which is added to the content of R_{src} to generate the effective address.

Solution : Flowchart

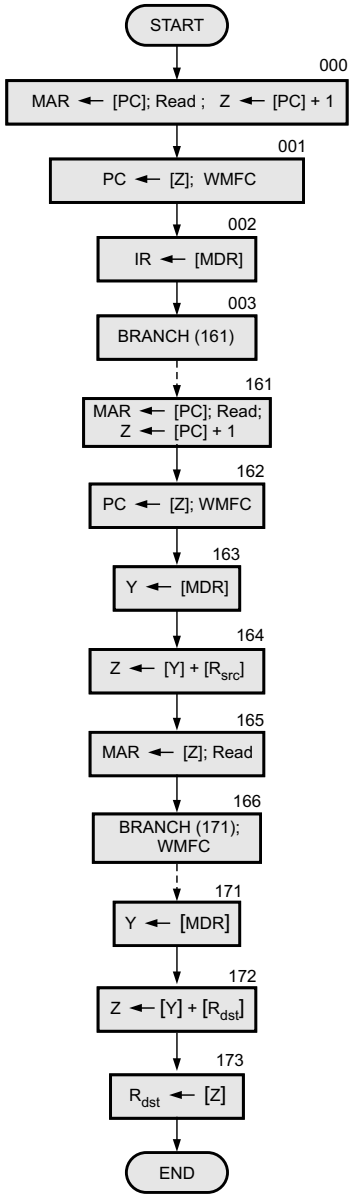


Fig. 3.4.13

Address (Octal)	Microinstruction
0 0 0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
0 0 1	Z_{out}, PC_{in}, MWFC
0 0 2	MDR_{out}, IR_{in}
0 0 3	$\mu Branch \left\{ \mu pc \leftarrow (101)_8 \text{ from instruction decoder}, \; \mu pc_{5,4} \leftarrow \left[IR_{10,9} \right], \; \mu pc_3 \leftarrow \left[\overline{IR}_{10} \right] \cdot \left[\overline{IR}_{9} \right] \cdot \left[IR_{8} \right] \right\}$ i.e. $\mu pc_{5,4,3} \leftarrow (110)_2$
1 6 1	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
1 6 2	Z_{out}, PC_{in}, MWFC
1 6 3	MDR_{out}, Y_{in}
1 6 4	R_{srcout}, SelectY, Add, Z_{in}
1 6 5	Z_{out}, MAR_{in}, Read
1 6 6	$\mu Branch \; \{ \mu pc \leftarrow (170)_8$ from instruction decoder $\mu pc_0 \leftarrow \overline{IR}_8 \}$; WMFC
⋮	
1 7 1	MDR_{out}, Y_{in}
1 7 2	R_{dstout}, SelectY, Add, Z_{in}
1 7 3	Z_{out}, R_{dstin}, End

Table 3.4.5 Microinstruction for Add const (R_{src}), R_{dst}

Example 3.4.8 *For a single bus organisation of CPU draw a flowchart and write microinstructions for instruction Add R_{src}, R_{dst}.*

Solution : Flowchart

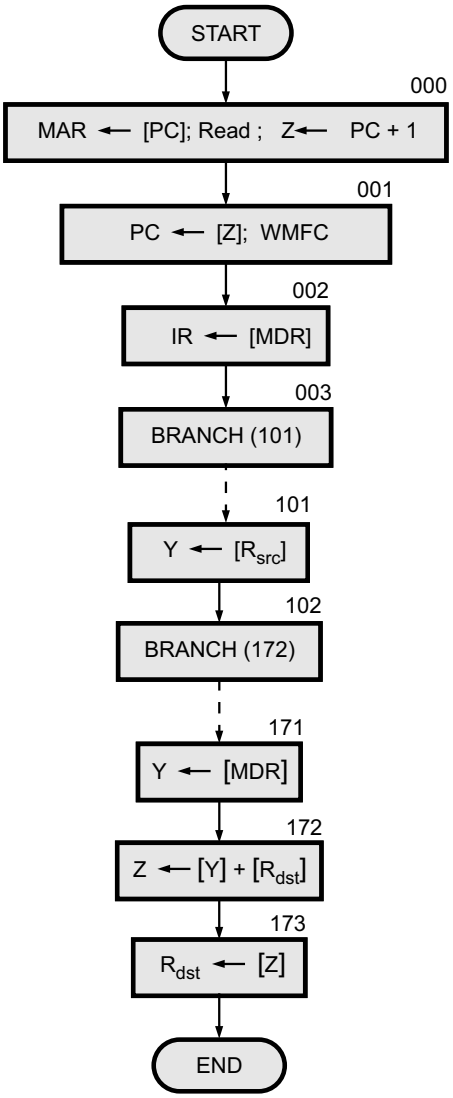


Fig. 3.4.14

Microinstructions

Address (Octal)	Microinstruction
0 0 0	PC_{out}, MAR_{in}, Read, Y_{in}, SelectC, Add, Z_{in}
0 0 1	Z_{out}, PC_{in}, MWFC

0 0 2	MDR_{out}, IR_{in}
0 0 3	$\mu Branch \left\{ \begin{array}{l} \mu pc \leftarrow (101)_8 \text{ from instruction decoder} \\ \mu pc_{5,4} \leftarrow \left[IR_{10,9} \right],\ \mu pc_3 \leftarrow \left[\overline{IR}_{10} \right] \cdot \left[\overline{IR}_{9} \right] \cdot \left[IR_8 \right] \end{array} \right\}$ i.e. $\mu pc_{5,4,3} \leftarrow (000)_2$
1 0 1	R_{srcout}, Y_{in}
1 0 2	$\mu Branch \ \{ \mu pc \leftarrow (170)_8$ from instruction decoder $\mu pc_0 \leftarrow \overline{IR}_8 \}$, WMFC
⋮	
1 7 1	MDR_{out}, Y_{in}
1 7 2	R_{dstout}, SelectY, Add, Z_{in}
1 7 3	Z_{out}, R_{dstin}, End

Table 3.4.6 Microinstruction for Add (R_{src}), R_{dst}

3.4.4 Microinstructions with Next Address Field

We have seen that in Fig. 3.4.5 requires several branch microinstructions. These microinstructions perform no useful operation in the datapath; they are needed only to determine the address of the next instruction. Thus they reduce the operating speed of the processor. This situation can become significantly worse when other microroutines are considered. This problem is solved by providing special address field for branch addresses as shown in the Fig. 3.4.15. Here, the branch address is stored in the special address field within the microinstruction. The branch address is loaded in CM address register when a branch condition is satisfied. The special address field within the microinstruction increases the size of the microinstruction. The size of the microinstruction can be reduced by storing part of the address (low order bits) in the microinstruction. This restricts the range of branch instructions to a small region of the CM and may therefore increase the difficulty of writing some microprograms.

(Fig. 3.4.15 see on next page)

3.4.5 Microinstruction Execution

A microprogrammed computer has two distinct levels of control.

i) The instruction level and ii) The microinstruction level.

At the instruction level, the CPU continuously executes instruction cycles that involve the following steps.

1. The CPU fetches an instruction from main memory, whose address is stored in the program counter PC.

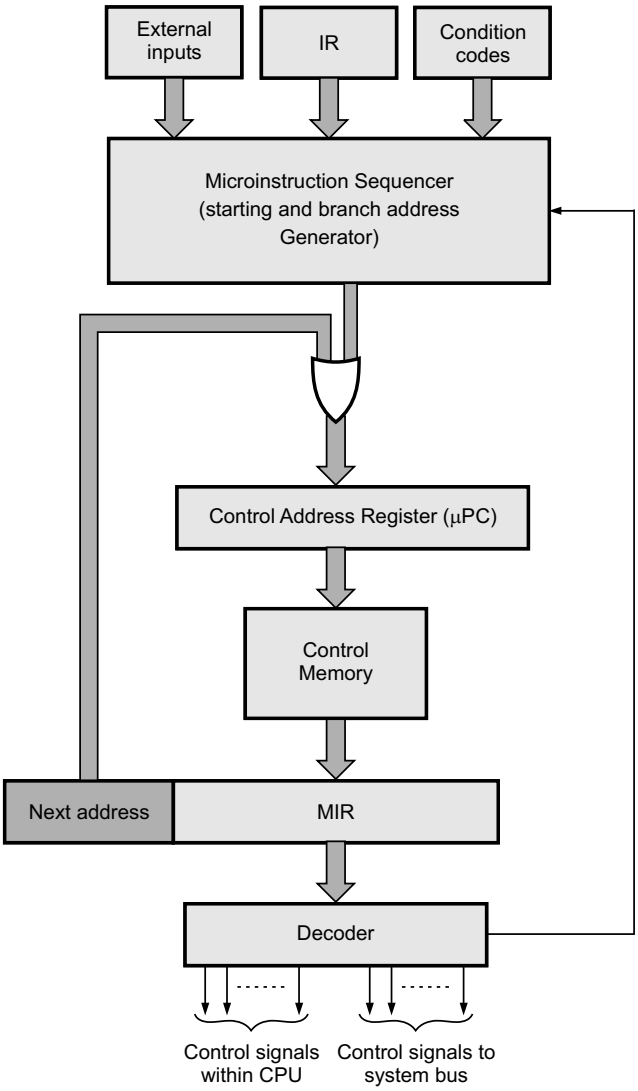


Fig. 3.4.15 Microinstruction sequencing organisation

2. The opcode part of I is placed in an instruction register IR, the operation specified by IR is then decoded and executed.

3. PC is altered to point to the next instruction to be fetched from M.

A similar sequence of operations takes place at the lower microinstruction level, where the control-unit continuously executes microinstruction cycles as follows :

1. The addressing portion (microprogram sequencer) of the control-unit fetches a microinstruction MI from the control memory CM, whose address is stored in the microprogram counter μpc.

2. MI is loaded into the microinstruction register MIR and is decoded to produce the required control signals.

3. μpc is altered to point the next microinstruction to be fetched from CM.

A microinstruction cycle can be executed faster than an instruction cycle, since microinstructions are stored within the CPU, whereas instructions must be fetched from an external memory. Microinstructions also normally require less decoding than instructions.

Review Questions

1. *Describe the organization of microprogrammed control unit.* **May-06, 07, 09, 10; Dec.-07, 12, Marks 4**
2. *With a neat diagram explain the basic organization of a microprogrammed control unit.* **Dec.-06, Marks 16**
3. *Explain microprogrammed control unit. What are the advantages and disadvantages of it ?* **May-08, Marks 16**
4. *State the advantages and disadvantages of microprogrammed control.* **Dec.-07, May-10**
5. *Explain the operation of this control unit with a typical set of microinstructions.* **Dec.-06, Marks 8**
6. *Compare horizontal and vertical format.* **Dec.-09, Marks 2**
7. *What modification is required in the basic organization to support microprogram branching ?* **May-09, Marks 4**
8. *Describe the organization of a typical micro-programmed control unit with the help of a neat diagram.* **May-11, Marks 8**
9. *Explain micro programmed control signal generation with a neat block diagram .* **May.-12, Marks 8**

3.5 Comparison between Hardwired and Microprogrammed Control

May-07, 12

Attribute	Hardwired control	Microprogrammed control
Speed	Fast	Slow
Control functions	Implemented in hardware.	Implemented in software.
Flexibility	Not flexible, to accommodate new system specifications or new instructions.	More flexible, to accommodate new system specification or new instructions redesign is required.
Ability to handle large/complex instruction sets	Somewhat difficult.	Easier.
Ability to support operating systems and diagnostic features	Very difficult (unless anticipated during design)	Easy

Design process	Somewhat complicated.	Orderly and systematic.
Applications	Mostly RISC microprocessors.	Mainframes, some microprocessors
Instruction set size	Usually under 100 instructions.	Usually over 100 instructions.
ROM size	—	2 K to 10 K by 20-400 bit microinstructions.
Chip area efficiency	Uses least area.	Uses more area.

Review Question

1. *State the differences between hardwired and microprogrammed control unit.*

May-07, 12, Marks 2

3.6 Bit Slicing and Bit Sliced Microprogram Sequencer

May-11

All processors have their structure and behavior fully defined during manufacture. System designer cannot change the processor architecture, data types and instructions, it can execute. Bit slicing allows the system designer to specify their own unique processor architecture.

The fundamental bit-sliced device is a processor slice, which is the Execution-unit for a simple CPU with a small data word size, typically 4-bits. The key property of a k-bit processor slice is the fact that n copies of the slice can be interconnected in a simple and regular manner to form an nk-bit processor that performs the same functions as a single slice on nk-bit rather than k-bit operands. The processor slices are interconnected in the form of a cascade circuit or one-dimensional array. The term "bit slicing" arises from the similarly between the processor array and a large processor that has been sliced into n identical subprocessors, each operating on a k-bit portion or slice of the data being processed by the array. As indicated in the Fig. 3.6.1, the same control lines are connected to all slices, so that, in general, they all perform the same operations at the same time; these operations are independent of n, the number of slices in the processor array. The k-bit data buses that transmit operands to or from the slices are simply merged to form nk-bit buses for the array.

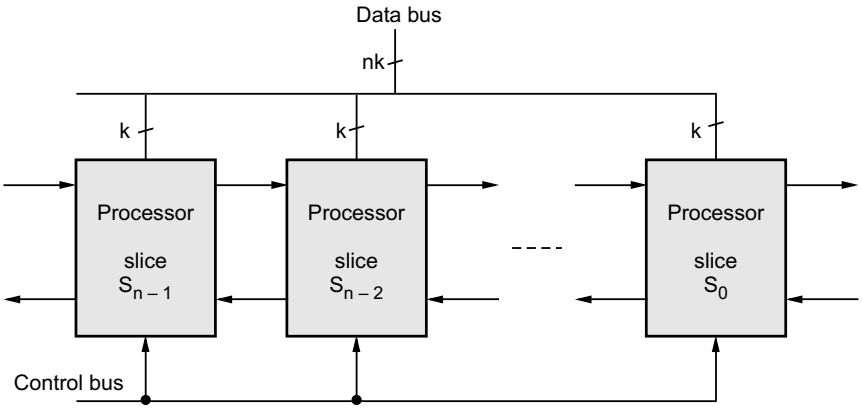


Fig. 3.6.1 General structure of nk-bit Bit sliced processor

3.6.1 Features of Bit Slicing

1. In a processor system if an instruction is not available for particular operation then that instruction is simulated by software. For example, it is possible to simulate multiplication instruction in 8085 microprocessor system by software. But this software simulation is much slower than hardware implementation. Bit slicing allows designer to implement any instruction by hardware.
2. Use of higher component density (MOS technologies) in most of the processor limits their speed of operation. Bit slicing solves these problems by providing flexible building blocks for processor design that can take advantage of the fastest current IC technologies. e.g. TTL, ECL etc.
3. In Bit slicing, the system designer can specify the processor word size(s), the composition of the instruction set, and other key architectural features, and can optimize them for a particular application.
4. A bit sliced processor can easily be partitioned dynamically into sub arrays of various sizes, which simplifies the processing of operands of different sizes. For example, 16 bit data words and 24 bit addresses.
5. The individual slices correspond to simple processor without the usual I-units. So they are small enough for fabrication using higher speed technologies like TTL and ECL.
6. The small word size used for the processor slices also solves another common problem in processors, the limited number of pins available in standard DIP packages.
7. The price paid for this increase in flexibility and speed is the large number of ICs needed to build a CPU, and the substantial increase in the total design effort required.

8. Non standard instruction set of bit sliced computers prevents use of widely available standard software.

Fig. 3.6.1 shows the internal organization of a bit sliced CPU.

The AMD 2900 series is one of the most widely used bit-sliced processor families. It comprises a large set of compatible ICs, all with 2900 part numbers, for constructing bit-sliced microprogrammable microcomputers. The series employs low power schottky TTL technology, which combines high speed with moderate power consumption. The members of the 2900 family may be grouped as follows.

1. Processor Slices.
2. Microprogram sequencers, some of which are bit-sliced.
3. Memories (RAMs and PROMs)
4. Miscellaneous SSI and MSI circuits (bus transceivers, decoders, registers etc.)

3.6.2 Processor Slice-2901

The first processor slice in the 2900 series is the 2901 4-bit processor, whose internal organization is as shown in the Fig. 3.6.1 (a). 2901 is a simple E unit. It has an eight function combinational ALU, 16 × 4 bit RAM, which constitutes a set of 16 general-purpose programmable registers and accumulator like result register denoted Q.

ALU : The ALU can perform addition and subtraction on 4-bit 2's complement numbers, as well as some conventional Boolean operations. It is an eight function combinational ALU. It consists of 4 flags : sign, overflow, zero, and carry. It uses these flags to store produced information of sign, zero, overflow and carry flag, which is required for ripple carry propagation and carry look ahead. The ALU can accept input from five different sources.

i) RAM [A]

ii) RAM [B]

iii) Q register

iv) Data bus D

v) Constant '0000'

The ALU has three different destinations

i) Y Bus

ii) Q register

iii) RAM [B] through RAM shifter

Source and destination are selected as specified by the instruction.

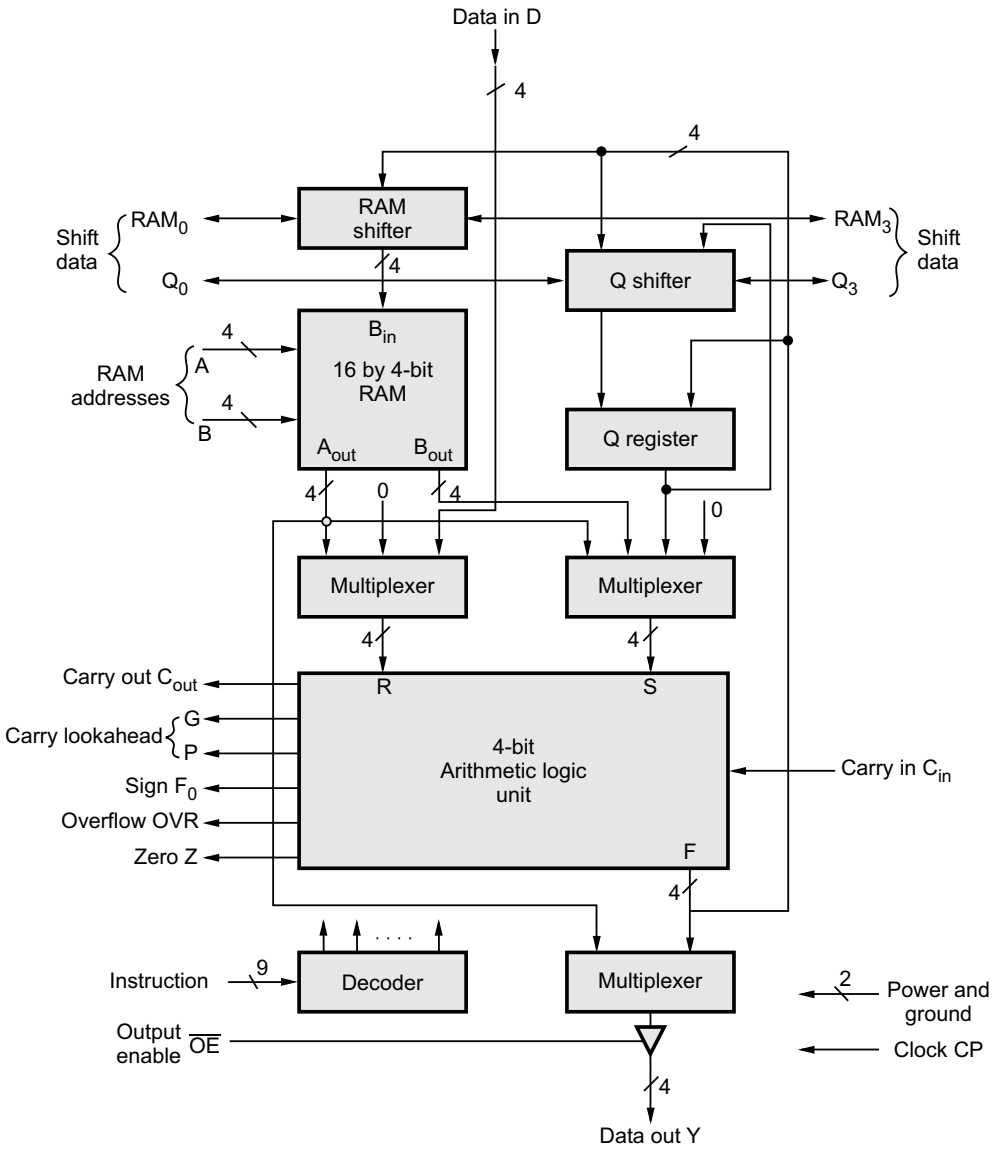


Fig. 3.6.1 (a) Internal organization of AMD 2901

Q shifter and RAM shifter : A pair of shift registers allows the contents of the Q register and the ALU result F to be shifted one position left or right.

RAM : The RAM has dual port i.e. two $\overline{RD}$s can be performed simultaneously. For this, it has 2 addresses A and B, corresponding to A and B Data output ports. A single input port to the RAM uses only B address. The RAM's two 4-bit addresses are obtained from a microinstruction control field supplied by I unit. Two 4 bit data buses D and Y allow

external data to be transferred to or from 2901 respectively. The combine bus is referred as **Instruction Bus**. This data on the 9 bit instruction bus determines the micro-operation to be performed by 2901.

The Instruction Bus has three major fields IS, IF, ID.

IS : Specifies sources of ALU's R and S input operands. Table gives the list of eight possible inputs and outputs.

IF : Specifies the function to be performed by ALU. Table gives the 8 different operations that ALU can perform.

ID : Indicates the destination for ALU's result and also specifies shifting operations for both F and Q.

a) IS

IS = I0 : I2	ALU sources	
	R	S
0 0 0	RAM [A]	Q
0 0 1	RAM [A]	RAM [B]
0 1 0	0	Q
0 1 1	0	RAM [B]
1 0 0	0	RAM [A]
1 0 1	D	RAM [A]
1 1 0	D	Q
1 1 1	D	0

b) IF

IF = I3 : I5	ALU function
0 0 0	$R + S + C_{in}$ (add)
0 0 1	$S - R - \overline{C_{in}}$ (subtract)
0 1 0	$R - S - \overline{C_{in}}$ (subtract)
0 1 1	$R \vee S$ (OR)
1 0 0	$R \wedge S$ (AND)
1 0 1	$\overline{R} \wedge s$ (Complement-AND)
1 1 0	$R \oplus S$ (Exclusive-OR)
1 1 1	$\overline{R \oplus S}$ (Exclusive-NOR)

c) ID

ID = I6 : I8	ALU Destination sand shift functions		
	Y	RAM [B]	Q
0 0 0	F	–	F
0 0 1	F	–	–
0 1 0	RAM [A]	F	–
0 1 1	F	F	–
1 0 0	F	F/2	Q/2
1 0 1	F	F/2	–
1 1 0	F	2F	2Q
1 1 1	F	2F	–

A complete microinstruction has six control fields as shown below.

Microinstruction format

A	B	IS	IF	ID	C_{in}
4 bit	4 bit	3 bit	3 bit	3 bit	1 bit

The microinstruction is a combination of six control fields = 18 bits microinstruction.

Actual control signals are generated by decoding 18-bit microinstruction with the help of internal decoders to get

A = 4 bits
B = 4 bits
IS = 8 bits
IF = 8 bits
ID = 8 bits
CM = 1 bit
Total = 33 bits

Control signals after internal decoding

3.6.3 Microprograms for 2901

a) Transfer logical complement of data on D bus to Q register

Logic : Complement function is not available in ALU. But this can be implemented with exclusive NOR (XNOR) function. If one input for XNOR function is 0 then output is complement of another input.

Steps : 1. R = D ; R input is D
2. S = 0 ; S input is 0

3. $F = \overline{R \oplus S}$; Result = $\overline{D}$

4. Q = F ; Route the result to Q

Now to implement this microinstruction format is as follows :

A = 0000	don't care
B = 0000	don't care
IS = 111	R = D S = 0
IF = 111	R ⊕ S
ID = 000	To route result to Q
C_{in} = 0	

b) Implement equation RAM [6] = (RAM [5] + RAM [6]) × 2

Microinstruction Format :

A = 0101	Contents of RAM [5]
B = 0110	Contents of RAM [6]
IS = 001	R = RAM [A] S = RAM [B]
IF = 000	$R + S + C_{in}$
ID = 111	Result in RAM [B] ← $(R + S + C_{in}) \times 2$
C_{in} = 0	

3.6.4 16-bit Bit Sliced Processor

A 4n-bit processor is formed by interconnecting n 2901s in the array as shown in the Fig. 3.6.2. A 16-bit processor composed of four 2901s. The 4-bit D and Y buses of the individual slices are merged to form 16-bit I/O buses for the array. Adjacent slices are linked in straight forward fashion by connecting their shift data lines and their carry-in and carry-out lines (C_{in} and C_{out}). The instruction bus I, the RAM address buses, and other input control lines are connected in parallel to all slices. The result of a 16-bit processor behaves as a 16-bit version of the 2901 with a 16-bit RAM, 16-bit data processing circuit, and 16-bit buses.

The 2901 generates four flags :

1. Zero flag 2. Carry flag

3. Sign flag 4. Overflow flag

The zero flag Z is set to 1 whenever 16-bit result is 0. To implement this zero flags of the individual 2901's are wired AND so that when all results 1 then only final zero flag is set to 1.

The sign and overflow flags are taken directly from the most significant slice. The carryout is connected to the carry in of the next 2901. Final carry is available at C_{out} of the most significant slice.

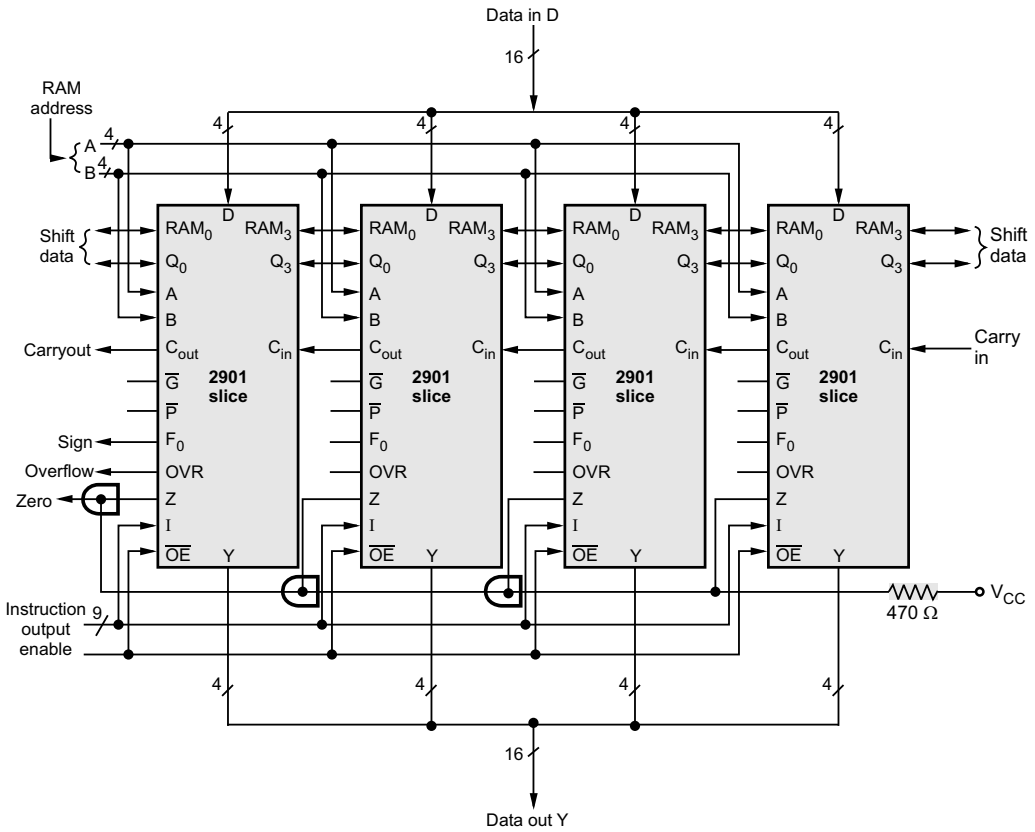


Fig. 3.6.2 16-bit Bit sliced processor

3.6.5 16-Bit Multiplication

Multiplication operation can be implemented in a bit-sliced processor via a short add and shift microprogram. The 16-bit Bit slice processor has 16-bit register, 16-bit RAM [A] and 16-bit RAM [B]. Using these three registers 16-bit multiplication can be performed.

The 16-bit Q register is used to store the multiplier operand, while the register RAM [A] stores the multiplicand and RAM [B] is used an accumulator register for partial products. RAM [B] and Q register form a single 32-bit right shift register. Fig. 3.6.3 shows the register organization for multiplication operation.

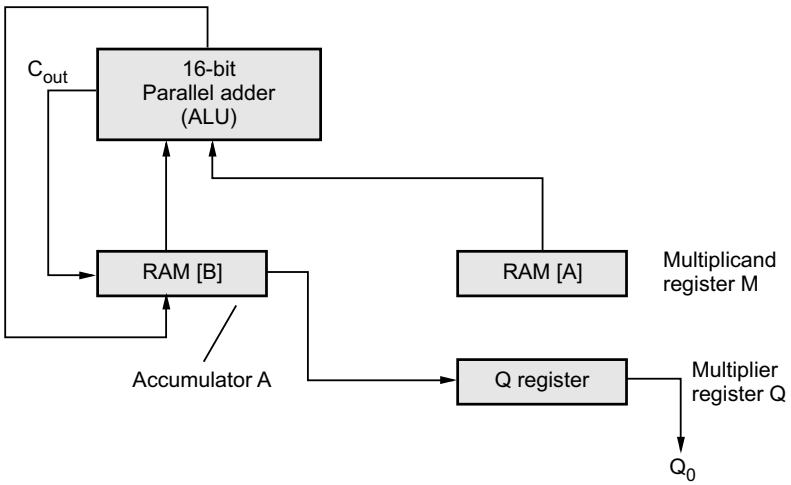


Fig. 3.6.3 Register organization for multiplication operation

Multiplication Operation Steps

i) Bit 0 of multiplier operand (Q register) is checked.

ii) If bit 0 is one then multiplicand and partial product are added and the result (32-bit) is shifted right by 1. i.e. RAM [A] + RAM [B] and shift right [RAM [B] + Q] by 1. If bit 0 is zero then [RAM [B] + Q] 32-bit contents are shifted right by 1.

iii) Steps i and ii are repeated 16 times to get desired result in RAM [B] (higher word) and Q register (lower word).

For simplicity these steps are explained for 4 × 4 bit multiplication.

Assume that multiplier operand is 1011 (Decimal 11) and multiplicand is 1010 (Decimal 10)

Then RAM [A] = 1010
RAM [B] = 0000 Initially Result = 0
Q register = 1011

Step 1 :

0 0 0 0	1 0 1 1

Bit 0 = 1

Step 2 :

Add RAM [A] contents into RAM [B] and then shift [RAM [B] + Q]
after addition

1 0 1 0	1 0 1 1

after shifting

0 →	0 1 0 1	0 1 0 1

Repeat steps 1 and 2.

Table 3.6.1 shows the intermediate results after each step in multiplication operation.

	0	0	0	0	1	0	1	**1**	Initial stage
bit 0 is 1 so add and shift	0	1	0	1	0	1	0	**1**	
bit 0 is 1 so add and shift	0	1	1	1	1	0	1	**0**	
bit 0 is 0 so only shift	1	1	0	1	1	1	0	**1**	
bit 0 is 1 so add & shift	0	1	1	0	1	1	1	0	Final result = 6E (Decimal 110)

Table 3.6.1

Review Questions

1. *What do you mean by bit-slicing ?*
2. *Explain the features of bit slicing.*
3. *Draw and explain the organisation of processor slice 2901.*
4. *Explain the organisation of 16-bit bit-sliced processor.*
5. *Explain the register organisation for 16-bit multiplication.*
6. *Describe various factors that reduce the performance of pipelined CPU and the mechanisms used to overcome it.* **May-11, Marks 16**

3.7 Applications of Microprogramming

The various microprogramming applications are :

Realization of Computers : Microprograms are used to implement the control unit of the computer.

Emulation : Microprograms are used in emulation process. Emulation refers to the use of a microprogram on one machine to execute programs originally written for another. The emulation process is used as an aid to users in migrating from one computer to another.

Operating System Support : Microprograms can be used to implement some of the primitives of operating system. This simplifies the task of operating system implementation. The implementation of primitives using microprogram also improves the performance of operating system.

High-Level Language Support : In high-level language various sub functions and data types can be implemented using microprogramming. This makes it easy to compile the program into an efficient machine language form.

Microdiagnostics : Microprogramming can be used to detection, isolation, monitoring and repair of system errors. These features are known as microdiagnotics and they significantly enhance the system maintenance facility.

User Tailoring : By implementing control memory using RAM it is possible to tailor the machine to the desired application.

Review Question

1. *Explain various applications of microprogramming.*

3.8 Pipeline Control

Dec.-03, 06, 07,08, 10, 11, 12, May-06, 07, 08, 09, 10, 12

The term pipelining refers to the temporal overlapping of processing. The concept of pipeline processing in a computer is similar to assembly lines in an industrial plant. Pipelines are nothing more than assembly lines in computing that can be used either for instruction processing or, in a more general sense, for performing any complex operation. To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. The concurrent execution of subtasks led to the tremendous improvement of system throughput in modern digital computer.

3.8.1 Principles of Pipelining

In pipelining, specialized **hardware stages** also known as **hardware segments** are linearly connected to perform a fixed function over a stream of data flowing from one end to the other. In advance computers, pipelines are applied for instruction execution, arithmetic computation, and memory accessing operations. A processor supporting such a hardware architecture is known as **pipeline processor**. Fig. 3.8.1 shows the basic structure of a k-stage pipeline.

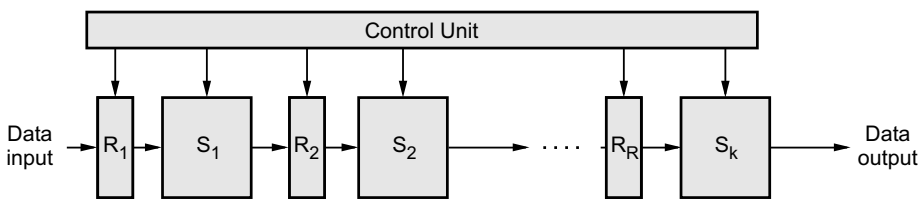


Fig. 3.8.1 Basic structure of a k-stage pipeline

As shown in the Fig. 3.8.1, a pipeline processor is constructed with k processing stages. Data inputs such as operands are fed into the pipeline at the first stage S_1. The processed results are passed from stage S_i to stage S_{i+1}, for all i = 1, 2, 3, … , k – 1. The final result emerges from the pipeline at the last stage S_k. A specific control mechanism is used to pass data flow along the pipeline stages. Depending on the control mechanism used we can categorize pipelines into :

- Asynchronous
- Synchronous

Example 3.8.1 *Perform the arithmetic operation (Ai * Bi) + (Ci * Di) with a stream of number. Specify a pipeline configuration to carry out the task. List the contents of all registers in the pipeline for i = 1 through 6.*

Solution : The Fig. 3.8.2 shows the pipeline configuration to carry out the given task. It consists of seven registers that receive new data with every clock pulse, two multipliers and one adder circuit. The suboperations performed in each stage of the pipeline are :

Stage 1 : R1 ← Ai, R2 ← Bi, R3 ← Ci, R4 ← Di

Stage 2 : R5 ← R1 * R2, R6 ← R3 * R4

Stage 3 : R7 ← R5 + R6

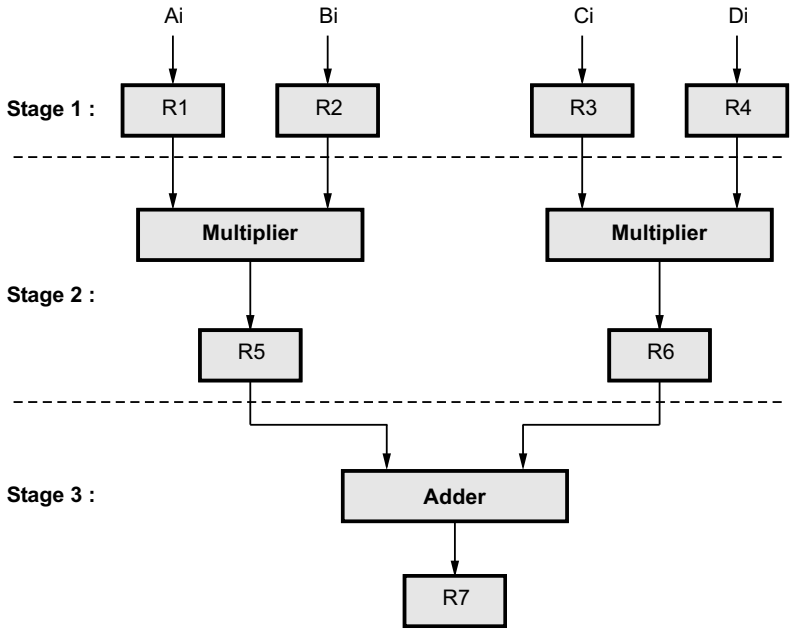


Fig. 3.8.2 Pipeline processing

The Table 3.8.1 shows contents of all registers in the pipeline for i = 1 through 6. The first clock pulse transfers A1, B1, C1 and D1 into R1, R2, R3 and R4, respectively. The second clock pulse transfers the product of R1 and R2 into R5 and product of R3 and R4 into R6. The same clock pulse transfers A2, B2, C2 and D2 into R1, R2, R3 and R4 respectively. The third clock pulse operates on all three stages simultaneously. It places A3, B3, C3 and D3 into R1, R2, R3 and R4, respectively, transfers the product of R1 and R2 into R5 and product of R3 and R4 into R6 and places sum of R5 and R6 into R7. It takes three clock pulses to fill up the pipe and retrieve the first output from R7. From there on, each clock produces a new output and moves the data one step down the pipeline. This is continued as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Clock Pulse Number (i)	Stage 1				Stage 2		Stage 3
	R1	R2	R3	R4	R5	R6	R7
1	A1	B1	C1	D1	-	-	-
2	A2	B2	C2	D2	A1 * B1	C1 * D1	-
3	A3	B3	C3	D3	A2 * B2	C2 * D2	A1 * B1 + C1 * D1
4	A4	B4	C4	D4	A3 * B3	C3 * D3	A2 * B2 + C2 * D2
5	A5	B5	C5	D5	A4 * B4	C4 * D4	A3 * B3 + C3 * D3
6	A6	B6	C6	D6	A5 * B5	C5 * D5	A4 * B4 + C4 * D4
7	-	-	-	-	A6 * B6	C6 * D6	A5 * B5 + C5 * D5
8	-	-	-	-	-	-	A6 * B6 + C6 * D6

Table 3.8.1 Contents of all registers in the pipeline for i = 1 through 6

3.8.2 Instruction Pipelines

We have seen various cycles involved in the instruction cycle. These fetch, decode and execute cycles for several instructions are performed simultaneously to reduce overall processing time. This process is referred to as **instruction pipelining**.

To apply the concept of instruction pipelining, we must subdivide instruction processing in number of stages as given below.

S_1 - Fetch (F) : Read instruction from the memory.

S_2 - Decode (D) : Decode the opcode and fetch source operand (s) if necessary.

S_3 - Execute (E) : Perform the operation specified by the instruction.

S_4 - Store (S) : Store the result in the destination.

Here, instruction processing is divided into four stages hence it is known as **four-stage instruction pipeline**. With this subdivision and assuming equal duration for each stage we can reduce the execution time for 4 instructions from 16 time units to 7 time units. This is illustrated in Fig. 3.8.3.

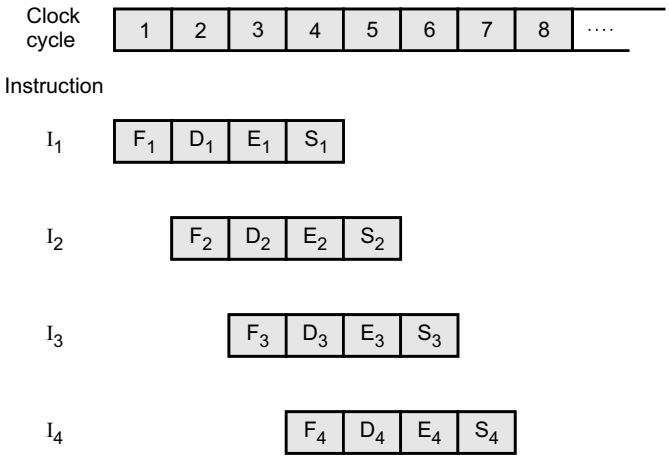


Fig. 3.8.3 Four stage instruction pipelining

In this instruction pipelining four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Fig. 3.8.4. These units are implemented such that they are capable of performing their tasks simultaneously and without interfering with one another. Information from the stage is passed to the next stage with the help of buffers.

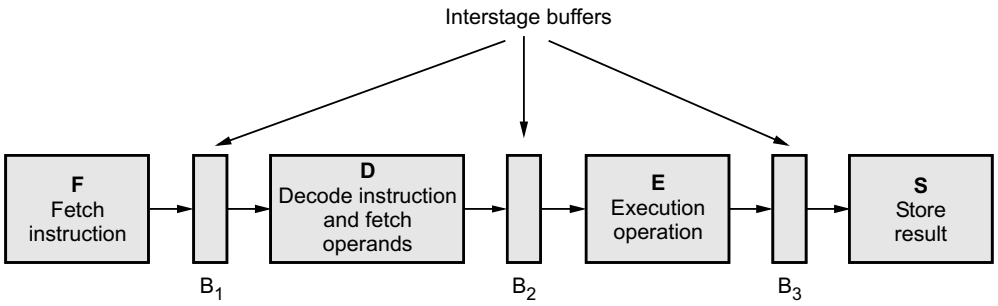


Fig. 3.8.4 Hardware organization for four-stage instruction pipeline

3.8.3 Implementation of Two-Stage Instruction Pipelining

The simplest instruction pipelining breaks instruction processing into two parts : A fetch stage S_1 and an execute stage S_2. When these two stages are overlapped, we get two-stage pipelining with increased throughput.

The Fig. 3.8.5 shows an implementation of a two-stage instruction pipeline. The fetch stage S_1 consists of the microprogram counter μPC, which is the source for microinstruction addresses, and the control memory CM, which stores the microinstructions. The execution stage S_2 consists of microinstruction register μIR, the decoders that extract control signals from the microinstructions in μIR, and the logic for determining next address or the branch address. The microinstruction register acts as a buffer register for stage S_2. With these two stages it is possible that, while instruction I_i with address A_i is being executed by stage S_2, the instruction I_{i+1} with the next

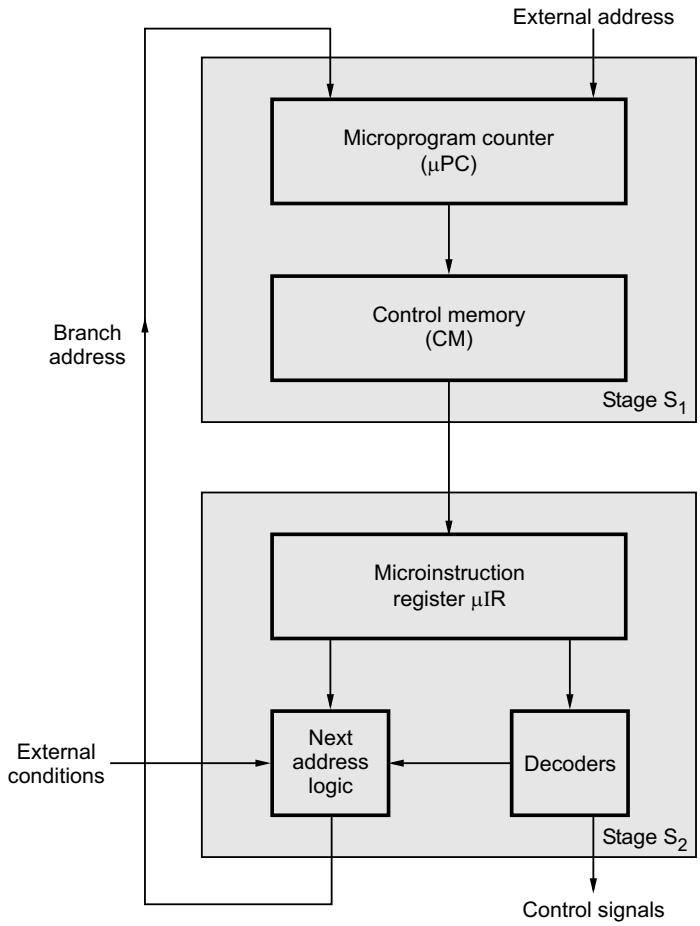


Fig. 3.8.5 Two-stage pipelined microprogram control unit

consecutive address A_{i+1} is fetched from memory by stage S_1. If on executing I_i in S_2 it is determined that a branch must be made to a nonconsecutive address, then the prefetched instruction I_{i+1} in S_1 has to be discarded. In such cases branch address is obtained directly from μI itself and fed back to S_1. The branch address is then loaded into μPC and next instruction is fetched from the branch address.

3.8.4 Implementation of Four-Stage Instruction Pipelining

The Fig. 3.8.6 shows the implementation of four-stage instruction pipelining. As shown in the Fig. 3.8.6 the CPU is directly connected to a cache memory, which is split into instruction and data parts, called the **I-cache** and **D-cache**, respectively. This splitting of the cache permits both an instruction word and a memory data word to be accessed in the same clock cycle. The four stages $S_1 : S_4$ shown in Fig. 3.8.6 perform the following functions :

S_1 : IF : Instruction fetching and decoding using the I cache.

S_2 : OL : Operand loading from the D-cache to register file.

S_3 : EX : Data processing using the ALU and register file.

S_4 : OS : Operand storing to the D-cache from register file.

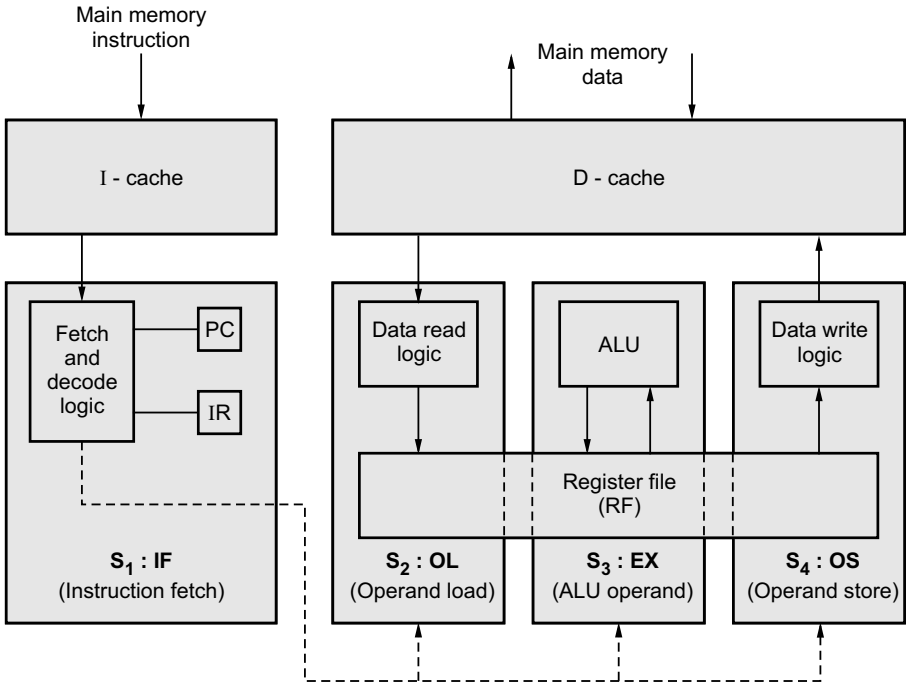


Fig. 3.8.6 Organization of CPU for four-stage instruction pipelining

Stages S_2 and S_4 implements memory load and store operations, respectively. Stages S_2, S_3 and S_4 share the CPU's local Register File (RF). The registers in the register file act as interstage buffer registers. The stage 3 implements data transfer and data processing operations of the register to register type using ALU of the CPU.

3.8.5 Pipeline Performance

To measure the performance of pipelined CPU we have to consider the performance measures. A pipeline's performance can be measured by its throughput in terms of millions of instructions executed per second or **MIPS**. Another popular measure of performance is the number of clock cycles per instruction or **CPI**. These quantities are related by the equation.

$$CPI = f/MIPS$$

where f is the pipeline's clock frequency in MHz, and the values of CPI and MIPS are average figures that can be determined experimentally by processing number of representative programs.

Performance Parameters of Pipelining

Each pipeline stage is a combinational logic circuit. It requires a specific amount of time in terms of propagation delay to process the input data. The precessing time in each stage is known as **stage delay**. It is denoted as t. The delays are also introduced due to interstage transfer data. These time delays are known as **interstage delay** and it is denoted as d.

Clock Cycle : To determine the time period for the clock cycle it is necessary to consider stage delay and interstage delay. The interstage delays can be same between the stages. However, stage delays may not be same for different stages. In such cases, maximum stage delay, denoted as t_m is considered to determine the time period for the clock cycle. The time period for clock cycle, t can be given as

$$t = \max_i \{t_i\}_1^k + d$$

$$= t_m + d$$

Usually, $t_m >> d$ and maximum stage delay dominates the clock period. Therefore, **pipeline frequency** which is inverse of clock period can be given as

$$F_p = \frac{1}{T_p}$$

Clock Skewing : Ideally, we expect the clock pulse to arrive at all stage registers at the same time. In practice, the same clock pulse may arrive at different stages with a time offset of s. This problem is known as **clock skewing**. The offset s is large for longer logic paths within the stages. Let us denote t_{max} be the time delay of the longest logic path within a stage and t_{min} be the time delay of the shortest logic path within a stage. These t_{max} and t_{min} put two constraints on the clock period as effect of clock skewing :

$$t_m \geq t_{max} + s \quad \text{and}$$

$$d \leq t_{min} - s$$

These two constraints on the clock period avoid a race in two successive stages. In ideal case s = 0, and hence $t_{max} = t_m$ and $t_{min} = d$.

Speedup, Efficiency and Throughput : Let us consider a space-time diagram in a pipeline processor. Fig. 3.8.7 (b) shows the space-time diagram of a four stage pipeline processor. As shown in the Fig. 3.8.7 (b), once the pipe is filled up, it outputs one result per clock period independent of the number of stages in the pipe. Ideally, a pipeline with k stages can process n tasks in $T_k = k + (n - 1)$ clock cycles, where k cycles are needed to complete the execution of the first task and the remaining n – 1 tasks require n – 1 cycles. Thus the total time required is

$$T_k = [k + (n - 1)] T_p$$

where T_p is the clock period.

The space-time diagram shown in Fig. 3.8.7 (b) has four stages and five tasks. Therefore, the ideal total time required is

$$T_k = [4 + (5 - 1)] T_p = 8 \text{ clock cycles (Refer Fig. 3.8.7 (b))}$$

The amount of time required to perform same number of tasks in a non-pipeline processor is nT_n.

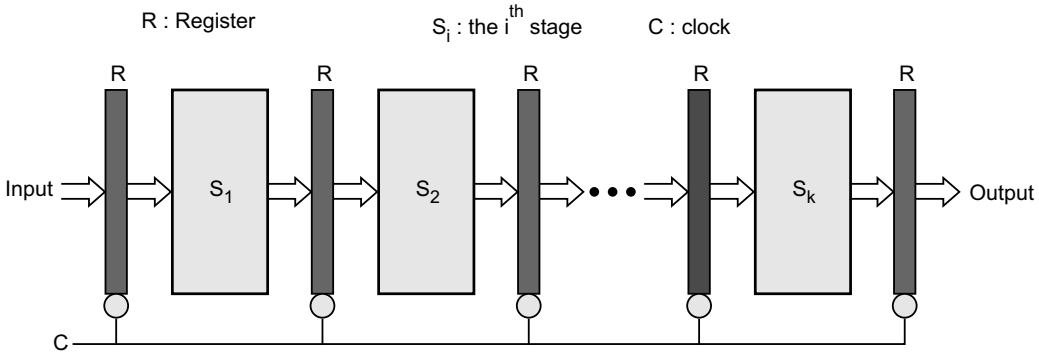


Fig. 3.8.7 (a) Basic structure of a pipeline processor

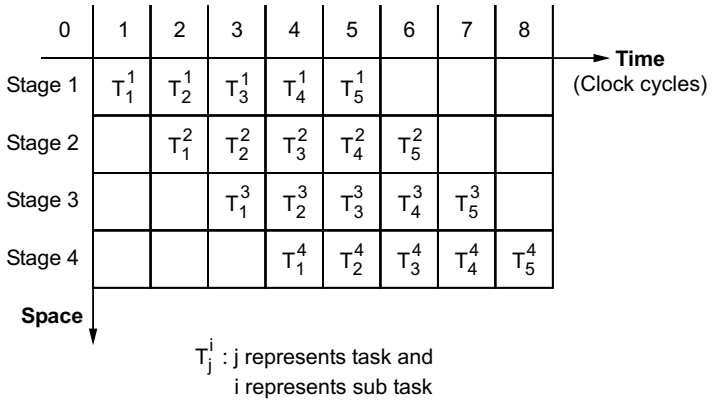


Fig. 3.8.7 (b) The space-time diagram showing the overlapped operations

Speedup Factor : The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is defined as

$$S_k = \frac{n\,T_n}{[k+(n-1)]\,T_p}$$

It can be noticed that maximum speedup is $S_k \rightarrow k$ as $n \rightarrow \infty$. However, this maximum speed is very difficult to achieve because of data dependencies between successive task (instructions), program branches, interrupts and other factors.

$$\underset{n\rightarrow\infty}{S_k} = \frac{nT_n}{nT_p} = \frac{T_n}{T_p}$$

If we assume that $T_n = kT_p$ we have

$$S_k = \frac{kT_p}{T_p} = k$$

Example 3.8.2 *A non-pipelined system takes 100 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 20 ns. Determine the speedup ratio of the pipeline for 200 tasks. What is the maximum speedup that can be achieved ?*

Solution : Given : $T_n = 100$ ns, $T_p = 20$ ns, $n = 200$, $k = 6$

$$\text{Speedup ratio } S_k = \frac{nT_n}{[k+(n-1)]\,t} = \frac{200\times100\times10^{-9}}{[6+(200-1)]\times20\times10^{-9}} = \mathbf{4.878}$$

$$\text{Maximum speedup} = \underset{n\rightarrow\infty}{\text{limit}} = \frac{nT_n}{[k+(n-1)]\,T_p} = \frac{nT_n}{n\,T_p}$$

$$= \frac{T_n}{T_p} = \frac{100\text{ ns}}{20\text{ ns}} = \mathbf{5}$$

Example 3.8.3 *Draw a space time diagram for a six segment stage pipeline showing the time it takes to process 8 tasks.*

Solution :

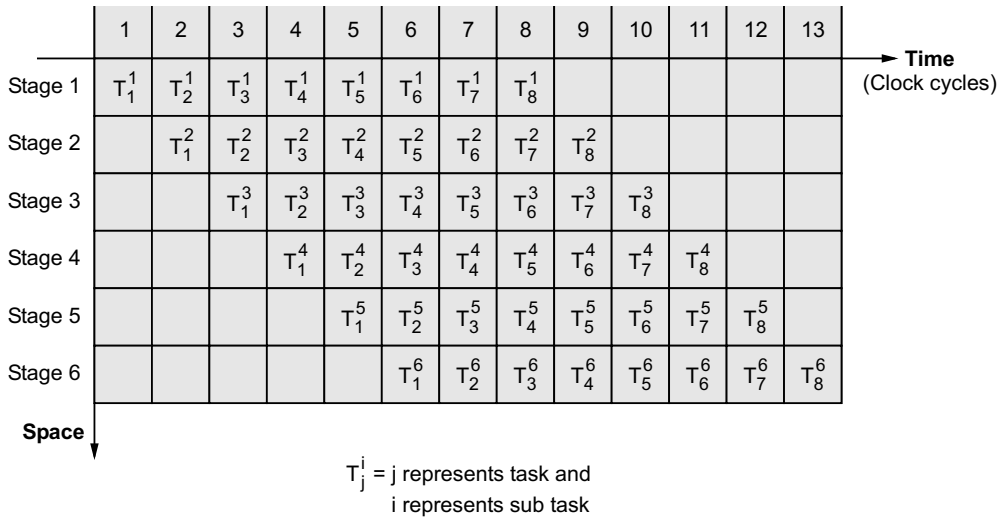

1 2 3 4 5 6 7 8 9 10 11 12 13
Time
(Clock cycles)
Stage 1 T_1^1 T_2^1 T_3^1 T_4^1 T_5^1 T_6^1 T_7^1 T_8^1
Stage 2 T_1^2 T_2^2 T_3^2 T_4^2 T_5^2 T_6^2 T_7^2 T_8^2
Stage 3 T_1^3 T_2^3 T_3^3 T_4^3 T_5^3 T_6^3 T_7^3 T_8^3
Stage 4 T_1^4 T_2^4 T_3^4 T_4^4 T_5^4 T_6^4 T_7^4 T_8^4
Stage 5 T_1^5 T_2^5 T_3^5 T_4^5 T_5^5 T_6^5 T_7^5 T_8^5
Stage 6 T_1^6 T_2^6 T_3^6 T_4^6 T_5^6 T_6^6 T_7^6 T_8^6
Space
T_j^i = j represents task and
i represents sub task

Fig. 3.8.8 The space time diagram

Efficiency : The efficiency of a linear pipeline is defined as a ratio of speedup factor and the number of stages in the pipeline. The efficiency of k-stage pipeline is given as

$$E_k = \frac{S_k}{k} = \frac{nk}{k+(n-1)} \div k$$

$$= \frac{n}{k+(n-1)}$$

For example, if speedup factor is 2.5 and number of stages are four, the efficiency can be given as

$$E_k = \frac{2.5}{4} = 0.625$$

It can be noticed that the efficiency approaches to unity when $n \to \infty$. When n = 1, efficiency is minimum. It is 1/k.

Throughput : The pipeline throughput H_k is defined as the number of results (tasks) that can be completed by a pipeline per unit time. It is given as

$$H_k = \frac{n}{[k+(n-1)]T_p}$$

$$= \frac{E_k}{T_p} \qquad \because E_k = \frac{n}{[k+(n-1)]}$$

$$= E_k \, f_p \qquad \because f_p = 1/T_p$$

In the ideal case, $H_k = 1/T_p = f_p$ when $E_k \to 1$. This means that the maximum throughput of a linear pipeline is equal to its frequency, which corresponds to one output result per clock period. The overall throughput of pipeline is always less than f_p. This is because usually $E_k < 1$.

Performance/cost ratio (PCR) : Let us consider t be the total time required for a non-pipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $\tau = t/k + d$, where d is the latch delay. Therefore maximum throughput for k-stage pipeline can be given as

$$f = \frac{1}{T_p} = \frac{1}{(t/k+d)}$$

The total pipeline cost is roughly estimated by c + kh, where c represents cost of all logic stages and h represents the cost of each latch. With this information, in 1973, Larson has defined a **pipeline performance/cost ratio** as

$$PCR = \frac{f}{c+kh} = \frac{1}{(t/k+d)(c+kh)}$$

Optimal number of stages : We know that performance of the pipeline can be increased by increasing pipeline stages. Many times pipeline stages are subdivided upto logic gates level called **micropipelining** to increase parallelism. However, increase in pipeline stages increases the cost of pipelining. Therefore, the optimal choice of the number of pipeline stages should be able to maximize a performance/cost ratio. Fig. 3.8.9 shows the graph of PCR Vs k-stages. The peak of curve corresponds to an optimal choice for the number of desired pipeline stages. It is given as

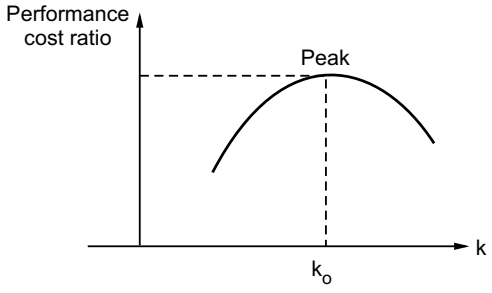


Fig. 3.8.9 Optimal number of pipeline stages

$$k_o = \sqrt{\frac{t.c}{d.h}}$$

where t is the total flow-through delay of the pipeline, c is the total stage cost, d is the latch delay and h is the latch cost.

3.8.6 Hazards in Instruction Pipelining

The timing diagram for instruction pipeline operation shown in Fig. 3.8.7 (b) completes the processing of one instruction in each clock cycle. This means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation shown in Fig. 3.8.7 (b) could be performed without any interruption throughout program execution. Unfortunately, this is not the case.

For many of reasons, one of the pipeline stages may not be able to complete its operation in the allotted time. For example, the stage E (Execute) is responsible for arithmetic and logic operations. In the timing diagram one clock period is allotted for this task. This stage can perform most of the operations in the allotted time. However, operations like divide may take more than one clock cycle and hence require more time to complete execution. The Fig. 3.8.10 shows an example in which the operation specified in instruction 2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the information in buffer B_2 must remain intact until the instruction execution stage has completed its operation. This means that stage 2 and in turn, stage 1 are blocked from accepting new instructions because the information in B_1 cannot be overwritten. Thus decode step for instruction and fetch step for instruction 5 must be postponed as shown in the Fig. 3.8.10.

The instruction pipeline shown in Fig. 3.8.10 is said to have been **stalled** for two clock cycles (clock cycles 5 and 6) and normal pipeline operation resumes in clock cycle 7. Any reason that causes the pipeline to stall is called a **hazard**.

Pipeline hazards are caused by resources-usage conflicts among various instructions in the pipeline. Such hazards are triggered by inter instruction dependencies. Such sequential instructions overlap their fetch, decode and execution through a pipeline processor, inter instruction dependencies may arise to prevent the sequential data flow in the pipeline. Until the completion of the previous instruction, the current instruction can not be initiated into the pipeline. Another type of hazards are due to a job scheduling problems. In all there are three types of hazards.

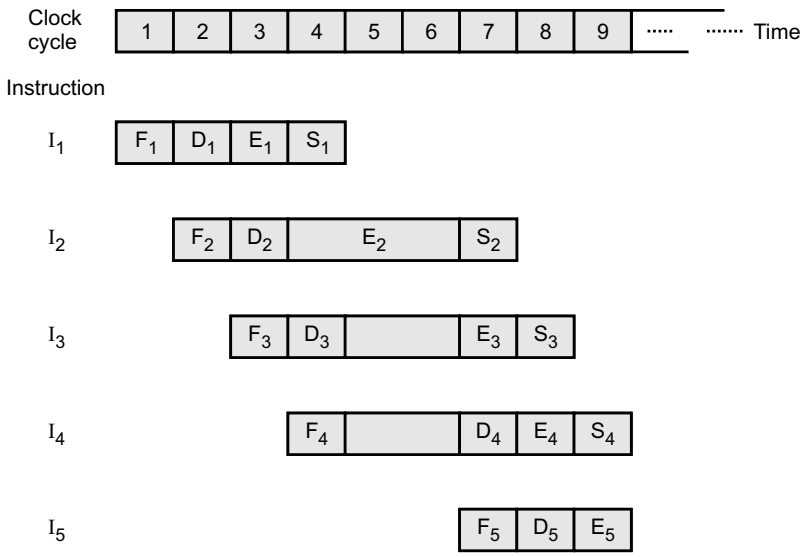


Fig. 3.8.10 Effect on pipeline of an execution operation taking more than one clock cycle

1. Structural hazards : These hazards are because of conflicts due to insufficient resources when even with all possible combination, it may not be possible to overlap the operation.

2. Data or Data dependent hazards : These result when instruction in the pipeline depends on the result of previous instructions which are still in pipeline and not completed.

3. Instruction or Control hazards : They arise while pipelining branch and other instructions that change the contents of program counter. The simplest way to handle these hazards is to stall the pipeline. Stalling of the pipeline allows few instructions to proceed to completion while stopping the execution of those which results in hazards.

3.8.6.1 Structural Hazards

The performance of pipelined processor depends on whether the functional units are pipelined and whether they are multiple execution units to allow all possible combination of instructions in the pipeline. If for some combination, pipeline has to be stalled to avoid the resource conflicts then there is a **structural hazard**. In other words, we can say that when two instructions require the use of a given hardware resource at the same time, the structural hazard occurs. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory for storage of the result while another instruction or operand needed is being fetched. If

instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. To avoid such type of structural hazards many processors use separate caches for instruction and data.

3.8.6.2 Data Hazards

When either the source or the destination operands of an instruction are not available at the time expected in the pipeline and as a result pipeline is stalled, we say such a situation is a **data hazard**.

Consider a program with two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of these two instructions can be performed concurrently. In such case the result of I_1 may not be available for the execution of I_2. If the result of I_2 is dependent on the result of I_1 we may get incorrect result if both are executed concurrently. For example, assume A = 10 in the following two operations :

$$I_1 : A \leftarrow A + 5$$

$$I_2 : B \leftarrow A \times 2$$

When these two operations are performed in the given order, one after the other, we get result = 30. But if they are performed concurrently, the value of A used in computing B would be the original value, 10, leading to an incorrect result. In this case data used in the I_2 depend on the result of I_1. The hazard due to such situation is called **data hazard** or **data dependent hazard**. To avoid incorrect results we have to execute dependent instructions one after the other (in-order). In other words we can say that, the read and write a shared variables by different instructions in a pipeline may lead to different results causing data-dependent hazards, if these instructions are executed out of order. The data dependent hazards can be classified according to various data update patterns :

- Write After Read (WAR) hazards,
- Read After Write (RAW) hazards and
- Write After Write (WAW) hazards

Fig. 3.8.7 shows these three hazards. As shown in the Fig. 3.8.7, there are two instructions 1 and 2, and it is assumed that instruction 2 should follow the instruction 1 according to program order. If the actual execution order of these two instructions violates the program order, incorrect results are read or written, thereby producing hazards.

A RAW hazard between the two instructions 1 and 2 may occur when instruction 2 attempts to read some data object that has been modified by instruction 1. A RAW hazard is also called **flow dependence hazard**. A WAR hazard may occur when instruction 2 attempts to modify some data object that is read by instruction 1. A WAR hazard is also called **antidependence hazard**. A WAW hazard may occur if both

instructions 1 and 2 attempt to modify the same data object. A WAW is also called **output dependence hazard**. These hazards must be prevented before these instructions enter the pipeline. This can be done by holding instruction 2 until the dependence on instruction 1 is resolved.

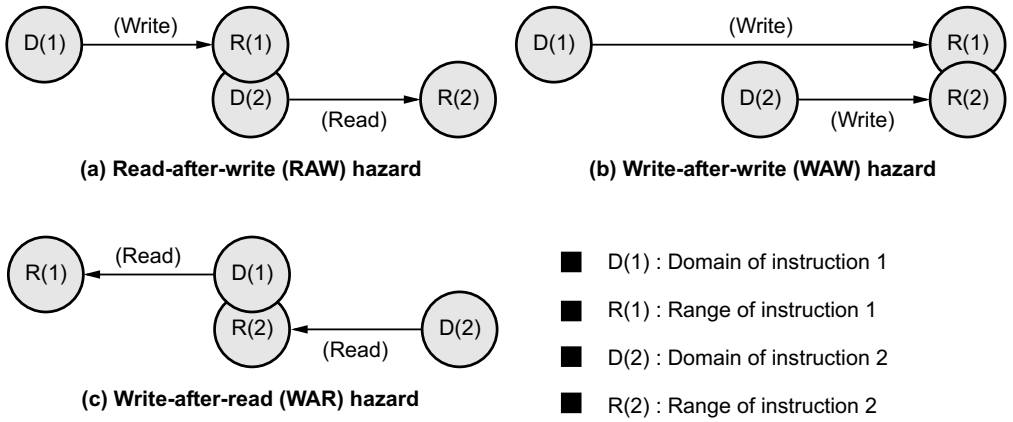


Fig. 3.8.11 Data-dependent hazards

As shown in the Fig. 3.8.11, D (1) represent domain of instruction 1 and R (1) represents range of instruction 1. The domain contains the input set (such as operands in the register or memory) to be used by instruction and the range corresponds to the output set of instruction. Using domains and ranges it is possible to derive conditions under which hazard can occur. These conditions are :

$$R(1) \cap D(2) \neq 0 \text{ for RAW hazard}$$

$$R(1) \cap R(2) \neq 0 \text{ for WAW hazard}$$

$$D(1) \cap R(2) \neq 0 \text{ for WAR hazard} \quad \ldots(3.8.1)$$

Hazard detection can be done in the instruction-fetch stage of a pipeline processor by comparing the domain and range of the incoming instruction with those of the instructions being processed in the pipeline, i.e. by checking the existence of any condition given in equation 1. A special hardware is used to check hazard conditions when instructions are being loaded into the prefetch buffer.

The Table 3.8.2 shows the possible hazards for various instruction types

Second instruction	First instruction			
	Arithmetic and load type	Store type	Branch type	Conditional branch type
Arithmetic and load type	RAW WAW WAR	RAW WAR	WAR	WAR
Store type	RAW	WAW		
	WAR			
Branch type	RAW		WAW	
Conditional branch type	RAW		WAW	

Table 3.8.2 Possible hazards for various instruction types

Techniques to avoid data hazards :

1. The easiest way to handle data hazards is to stall the pipeline. Pipeline interlocks detect the hazard and stalls the pipeline beginning with the instruction that requires the data till it is made available by earlier instruction. This delay or stall called a **bubble**.
2. The second simple hardware technique which can handle data hazard is called **forwarding** or **register by passing**. In this technique, ALU results are fed back as ALU inputs. When the forwarding logic detects the previous ALU operation has the operand for current instruction, it forwards ALU output instead of register file.
3. In some cases register forwarding technique also fails leading to pipeline stall.

Example :

LW	R1, 60(R5)	IF	ID	EX	MEM	WB		
ADD	R4, R1, R8	IF	ID	EX	MEM	WB		
OR	R7, R3, R8		IF	ID	EX	MEM	WB	
XOR	R5, R2, R7			IF	ID	EX	MEM	WB

Here LW instruction does not have result ready till the memory access is over. While ADD requires it during execution (EX) stage. The simplest way to handle such situation is to stall ADD instruction for one clock. But it will degrade the performance.

Rather than allowing the pipeline to stall, the compiler can rearrange above instructions to avoid a read following a load. It is used in RISC processors. It is called **pipeline scheduling**.

After scheduling :

```
LW    R1, 60(R5)
OR    R7, R3, R8
ADD   R4, R1, R8
XOR   R5, R2, R7
```

A load which requires that following instruction does not use its result is said to be **delayed load** and the pipeline slot after load instruction is called **delayed slot**. Some machines like MIPS 2000 rely heavily on software scheduling to avoid data hazard. In these machines there is no support in hardware to detect and stall the pipeline. However, the compiler can insert number of NOP instructions to fill the delayed slot.

3.8.6.3 Instruction or Control Hazards

These are caused by change in the relative execution timing of the overlapped instructions due to pipelining. Control hazards can cause greater degradation in performance than data hazards. These occur due to instruction like branch or jump which change flow of instruction which changes content of PC.

Branch instruction (i)	IF	ID	EX	MEM	WB					
i + 1	IF	Stall	Stall	IF	ID		EX	MEM		WB
i + 2		Stall	Stall	Stall	IF		ID	EX		MEM
i + 3			Stall	Stall	Stall	IF	ID	EX		
i + 4			Stall	Stall	Stall	IF	ID			
i + 5					Stall	Stall		Stall		IF

Fig. 3.8.12

If branch is taken as shown in Fig. 3.74 then the PC value is not known till the end of MEM. Next three instructions are fetched even though they are not required. Hence they have to be flushed after branch is taken and new set of instructions have to be fetched from the branch address. In simple pipelines there is no problem of flushing but in complex pipelines, flushing is not simple. One way to handle it is to stall the pipeline after instruction decoded is branch.

For machine with lower CPI even this loss of two cycles due to branching is significant. To tackle it, prediction of whether branch is taken or not early in pipeline has to be done.

Modified pipeline for branch instruction

IF	$MAR \leftarrow PC$	
	$IR \leftarrow MEM\ [MAR]$	
	$PC \leftarrow PC + 4$	
ID	$A \leftarrow RS_1,\ B \leftarrow RS_2$	Target calculation
	$BTA \leftarrow PC + Branch\ offset$	done in ID stage
	If cond THEN $PC \leftarrow BTA$	so branch penality only 1

EX	–	Not used
MEM	–	
WB	–	

Branch Penalty

The penalty due to branches depends upon two factors :

- Instruction set : In RISC penalty is less than complex instruction set computers.
- Depth of pipeline : More deep pipelining more is branch penalty.

The branch penalties can be reduced by proper scheduling through compiler techniques. The basic idea behind these techniques is to fill the 'delay slot' with some useful instruction which in most cases will be executed.

Delayed Branching

There are three ways to fill the delay slot :

1. The delay slot is filled with an independent instruction before branch. In this case performance always improves.
2. The delay slot is filled from the target branch instructions. Performance improves if only branch is taken.
3. The delay slot is filled with an instruction which is one of the fall through instruction. In this case performance improves if branch is not taken.

These above techniques are called **delayed branching**.

Conditional Branching

The conditional branching is a major factor that affects the performance of instruction pipelining. There are several approaches to deal with conditional branching.

These are :

- Multiple Streams
- Prefetch Branch Target
- Loop Buffer
- Branch Prediction

Multiple Streams

We know that, a simple pipeline suffers a penalty for a branch instruction. To avoid this, in this approach they use two streams to store the fetched instruction. One stream stores instructions after the conditional branch instruction and it is used when branch condition is not valid. The other stream stores the instruction from the branch address and it is used when branch condition is valid.

Drawbacks :

- Due to multiple streams there are contention delays for access to registers and to memory.
- Each branch instruction needs additional stream.

Prefetch Branch Target

In this approach, when a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This already fetched target is used when branch is valid.

Loop Buffer

A loop buffer is a small very high speed memory. It is used to store recently prefetched instructions in sequence. If conditional branch is valid, the hardware first checks whether the branch target is within the buffer. If so, the next instructions are fetched from the buffer, instead of memory avoiding memory access.

Advantages

- In case of conditional branch, instructions are fetched from the buffer saving memory access time. This is useful for **loop** sequences.
- If a branch occurs to a target just a few locations ahead of the address of the branch instruction, we may find target in the buffer. This is useful for IF-THEN-ELSE-ENDIF sequences.

Branch Prediction

Prediction techniques can be used to check whether a branch will be valid or not valid. The common prediction techniques are :

- Predict Never Taken
- Predict Always Taken
- Predict By Opcode
- Taken/Not Taken Switch
- Branch History Table

The first three approaches are simple. In the first two approaches if prediction is wrong a page fault or protection violation error occurs. The processor then halts prefetching and fetches the instruction from the desired address.

In the third prediction technique, the prediction decision is based on the opcode of the branch instruction. The processor assumes that the branch will be taken from certain branch opcodes and not for others.

The fourth and fifth prediction techniques are dynamic ; they depend on the execution history of the previously executed conditional branch instruction.

In Taken/Not taken switch system one or two tag bits are associated with each conditional branch instruction. These bits reflect the recent history of the conditional branch instruction. These are referred to as a taken/not taken switch, that directs the processor to make a particular decision the next time the instruction is encountered. For example, in a single bit tag, 1 is recorded if the last execution of this instruction resulted valid branch and 0 is recorded for no branch. When this instruction is fetched again, fetching unit checks the tag bit. If tag bit is 0 it fetches next instructions in the sequence ; otherwise it fetches instructions from the branch address.

This technique is very useful for loop sequences. In loop sequences N–1 times branching is taken. Therefore, the prediction is wrong twice for each loop : Once on entering the loop and once on exiting.

The use of history bits just explained, has one drawback : If the decision is made to take the branch, the target instruction cannot be fetched until the branch instruction is decoded. Greater performance could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, it is necessary to store branch address alongwith the tag bits. In the branch history table, technique, the branch address alongwith the tag bits is stored in the branch target buffer or branch history table.

3.8.7 Influence on Instruction Sets

There is an influence of instructions on the pipelining. Some instructions are much better suited to pipelined execution than others. In this section we will discuss, how the features of the instruction affect the pipelined execution. Out of the different aspects, two important of them, addressing modes and condition code flags are explained below.

3.8.7.1 Addressing Modes

Many processors provide various combinations of addressing modes such as index, indirect, autoincrement, autodecrement, and so on. We can broadly classify these addressing modes as simple addressing modes and complex addressing modes. A complex address mode may require several accesses to the memory to reach the named operand while simple addressing mode requires only one access to the memory to reach the named operand.

Let us see the equivalent operations using complex and simple addressing modes. Consider the instruction Load R_1, (X (R_0)). This instruction has a complex addressing mode because this instruction requires two memory accesses one to read location $X + [R_0]$ and then to read location $[X + [R_0]]$. Refer Fig. 3.8.13. As shown in the Fig. 3.8.13, if R_1 is a source operand in the next instruction, that instruction has to be stalled for two cycles.

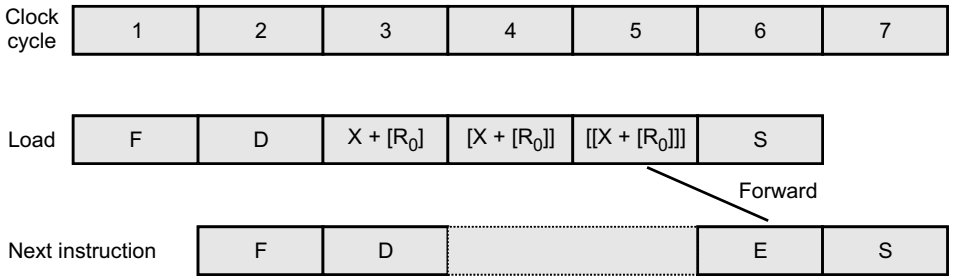


Fig. 3.8.13 Operation using complex addressing mode

If we want to perform the same operation using simple addressing mode instructions, we require to execute three instructions as shown below :

ADD R_1, R_0, # X

Load R_1, (R_1)

Load R_1, (R_1)

The ADD instruction performs the operation $R_1 \leftarrow [R_0] + X$. The two load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the single load instruction having a complex addressing mode. This is illustrated in Fig. 3.8.14.

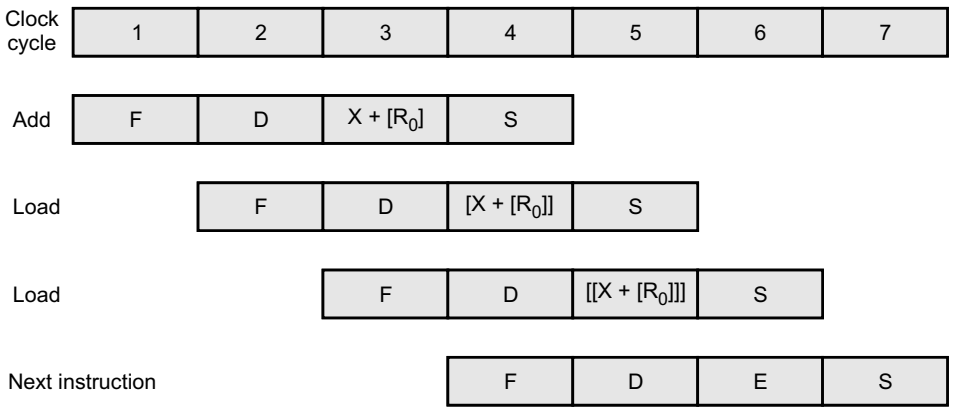


Fig. 3.8.14 Operation using simple addressing mode

The above example indicates that, in a pipeline processor, complex addressing mode do not necessarily lead to faster execution. But it reduces the number of instructions needed to perform particular task and hence reduces the program space needed in the main memory. However, complex addressing mode instructions have following disadvantages :

1. Long execution times of complex addressing mode instructions may cause pipeline to stall.
2. They require more complex hardware to decode and execute them.
3. They are not convenient for compilers to work with.

Due to above reasons the complex addressing mode instructions are not suitable for pipelined execution. Therefore, these addressing modes are avoided in modern processors. The addressing modes of modern processors have following features :

1. They access operand from the memory in only one cycle.
2. Only load and store instructions are provided to access memory.
3. The addressing modes used do not have side effects. (When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a **side effect**).

The addressing modes such as register, register indirect, and index have above listed features. The register and register indirect addressing modes do not require any address computation, and the index addressing mode the address can be computed in only one cycle.

3.8.7.2 Condition Codes

Most of the processors store condition code flags in their status register. The conditional branch instruction checks the condition code flag and the flow of program execution is decided. When the data dependency or branch exists, the pipeline may stall. So, for a pipelined processor, a special optimizing compiler is designed which reorders the instructions in the program which is to be executed. This avoids the stalling the pipeline of course, the compiler takes the care to obtain the correct output while reordering the instructions.

The following two important points should be kept in mind to handle condition codes.

a. The condition code flags should be affected by as few instructions as possible. This provides flexibility in reordering instructions. Otherwise, the dependency produced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

b. It should be known to compiler that, in which instructions of a program the condition codes are affected and in which they remain unaffected.

This desired flexibility is provided when the instruction set is designed with pipelining in mind.

3.8.8 Branch Prediction

There are two types of branch prediction strategies :

- Static branch strategy
- Dynamic branch strategy

Static Branch Strategy : In this strategy branch can be predicted based on branch code types statically. This means that the probability of branch with respect to a particular branch instruction type is used to predict the branch. This branch strategy may not produce accurate results every time.

Dynamic Branch Strategy : This strategy uses recent branch history during program execution to predict whether or not the branch will be taken next time when it occurs. It uses recent branch information to predict the next branch. The recent branch information includes branch prediction statistics such as :

T : Branch taken

N : Not taken

NN : Last two branches not taken

NT : Not branch taken and previous takes

TT : Both last two branch taken

TN : Last branch taken and previous not taken

This information is stored in the buffer called **Branch Target Buffer** (BTB). Along with above information branch target buffer also stores the address of branch target. Fig. 3.8.15 shows the organization of branch target buffer.

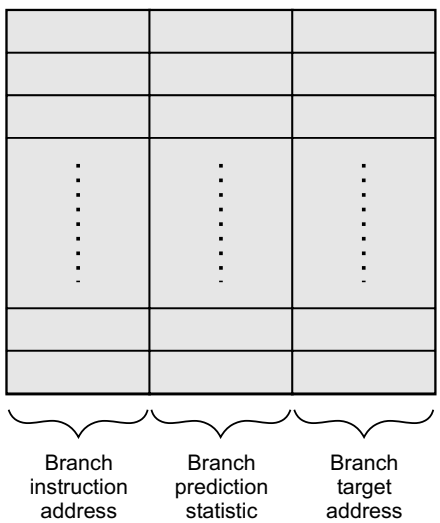
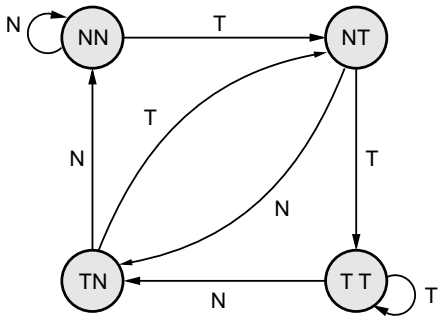


Fig. 3.8.15 Branch target buffer organization

Fig. 3.8.16 shows a typical state diagram used in dynamic branch prediction. This state diagram allows the backtracking of last two instructions in a given program. The branch target buffer entry contains the backtracking information which guides the prediction. The prediction information is updated upon completion of the current branch.



T : Branch taken
N : Branch not taken
NN : Last two branches not taken
NT : Not branch taken and previous taken
T T : Last two branches taken
TN : Last branch taken and previously not taken

Fig. 3.8.16 A typical state diagram used in dynamic branch prediction

To make branch overhead zero, the branch target buffer is extended to store the target instruction itself and a few of its successor instructions. This allows processing of conditional branches with zero delay.

Review Questions

1. *What do you mean by instruction pipelining ?*
2. *List the factors affecting the performance of instruction pipelining.*
3. *How does conditional branch instruction affect the performance of instruction pipelining ? Explain with the help of timing diagram.*
4. *Explain the various approaches used to deal with conditional branching.*
5. *Draw and explain the implementation of two stage instruction pipelining.*
6. *Draw and explain the implementation of four stage instruction pipelining.*
7. *Write a short note on pipeline performance.*
8. *Define pipeline's performance/cost ratio and speed up performance.* **Dec.-03, Marks 2**
9. *How do control instructions like branch, cause problems in a pipelined processor ?* **May-06, Marks 2**
10. *Explain how pipelining helps to speed-up the processor. Discuss the hazards that have to be taken care of in a pipelined processor.* **May-06, Marks 12**
11. *Explain the basic concepts of pipelining and comparing it with sequential processing. Draw needed diagrams.* **Dec.-06, Marks 8**
12. *What is the ideal speed-up expected in a pipelined architecture with 'n' stages ? Justify your answer.* **May-07, Marks 2**

13. *Discuss the various hazards that might arise in a pipeline. What are the remedies commonly adopted to overcome/minimize these hazards ?* **May-07, Marks 16**
14. *What is data hazard in pipelining? What are the solutions?* **Dec.-07, Marks 2**
15. *Design a 4-stage instruction pipeline and show how its performance is improved over sequential execution.* **Dec.-07, Marks 8**
16. *Highlight the solutions of instruction hazards.* **Dec.-07, Marks 8**
17. *Draw the structure of two stage instruction pipeline.* **Dec.-07, Marks 2**
18. *What is pipelining and what are the advantages of pipelining?* **May-08, Marks 2**
19. *What is branch hazard ? Describe the methods for dealing with the branch hazards.* **May-08, Marks 10**
20. *What is data hazard? Explain the methods for dealing with the data hazards.* **May-08, Marks 16**
21. *Define data hazards.* **Dec.-08, Marks 2**
22. *Explain the concept of pipelining. Discuss the influence of various pipelining hazards on instruction set.* **Dec.-08, Marks 16**
23. *Explain structural hazard.* **Dec.-08, Marks 2**
24. *List the conditions to overcome data hazard.* **Dec.-08, Marks 2**
25. *Discuss in detail about hazards due to unconditional branching.* **Dec.-08, Marks 10**
26. *What are the three types of hazards that cause performance degradation in pipelined processors ? Explain them in detail.* **May-09, Marks 8**
27. *Distinguish between static and dynamic branch prediction approaches.* **May-09, Marks 2**
28. *Define pipelining.* **May-09, Marks 2**
29. *Explain the various pipelining hazards and their remedies in the processor.* **May-09, Marks 16**
30. *Explain the various types of pipeline hazards and their resolution mechanisms in detail.* **May-10, Marks 16**
31. *What is a pipeline control?* **Dec.-10, Marks 2**
32. *Explain the different type of hazards that can occur in a pipeline.* **Dec.-10, Marks 16**
33. *What do you mean by hazard ? Explain the different types of hazards with the way and means of solving them.* **May-12, Dec.-12, Marks 16**

3.9 Superscaling Processing **Dec.-11, 12**

Microprocessors such as the PowerPC reach performance levels greater than one instruction per cycle by fetching, decoding and executing several instructions concurrently. This mode of operation is called **superscaler**. A processor capable of parallel instruction execution and having performance level greater than one instruction per cycle is known as **superscaler processor**. A superscaler processor has multiple execution units (E-units), each of which is usually pipelined, so that they constitute a set

of independent instruction pipelines. Its Program Control Unit (PCU) is capable of fetching and decoding several instructions concurrently. It can issue up to k instructions simultaneously to the various E-units. Where k is called **instruction issue degree**. It can be six or more using current technology.

To fetch, decode and execute several instruction concurrently, the superscaler operation requires processor to detect and exploit instruction-level parallelism hidden in the programs it executes. It has to handle following important issues :

- **Instruction Type :** A floating-point add instruction has to be issued to a floating point E-unit and not to an integer E-unit.
- **E-unit availability :** An instruction can be issued to a pipelined E-unit only if no collision will result, as determined by the pipeline's reservation table.
- True data dependency
- Procedural dependency
- Resource conflicts
- Output dependency
- Antidependency

True Data Dependency

Let us see the following instructions :

Sub r_1, r_2 ; Subtract the contents of r_2 from r_1 and store the result in r_1

Move r_3, r_1 ; Load register r_3 with the contents of r_1

In the above instruction sequence, the second instruction can be fetched and decoded, but cannot be executed until the first instruction executes. The reason is that the second instruction requires the data produced by the first instruction. This situation is referred to as a **true data dependency**. This limits the performance of any type of pipeline.

Procedural Dependency

We have seen earlier that the branch instructions complicate the pipeline operation. There are branch prediction techniques, but they cannot improve performance hundred percent. In branch prediction technique, **taken or not taken**, we cannot execute instruction until branch is executed. This situation is referred to as **procedural dependency**.

Resource Conflict

While executing number of instructions in parallel, the situation may occur that two or more instructions require same resource at the same time. For example, two or more instructions may require references to memories, caches, buses register-file ports, and functional units at the same time. This situation is referred to as **resource conflict**.

Output Dependency

Let us see the following instructions.

R3 = R3 + R5 ← I1
R4 = R3 + 1 ← I2
R3 = R5 + 1 ← I3
R6 = R3 + R4 ← I4

In the above instruction sequence we cannot execute instruction I2 before I1 because of the true data dependency and for similar reasons we also cannot execute instruction I4 before I3. But there is one more dependency which exists between instruction I3 and instruction I1. We cannot execute instruction I3 before instruction I1, because if I3 executes prior to completion of I1, then wrong value of the contents of R3 will be fetched for the execution of instructions I1 and I4. This situation is referred to as **output dependency**.

Antidependency

Let us see the following instructions

R3 = R3 + R5 ← I1
R4 = R3 + 1 ← I2
R3 = R5 + 1 ← I3
R6 = R3 + R4 ← I4

In the above instruction sequence instruction I3 cannot be completed its execution before instruction I2 begins execution and had fetched its operands. This is so because instruction I3 updates register R3, which is a source operand for instruction I2. This situation is referred to as **antidependency**. Here, constraints are exactly reverse that of true data dependency.

Control dependencies : To maintain high performance levels, techniques are needed to reduce the impact of branch instructions on pipeline efficiency.

Program order : Instructions must eventually produce results in the order specified by the program being executed. The result may, however, be computed out-of-order internally to improve the CPU's performance.

3.9.1 Instruction Level Parallelism and Machine Parallelism

When instructions in a sequence are independent and can be executed in parallel by overlapping, there is an instruction level parallelism. On the other hand, machine parallelism is a measure of the ability of the processor to take advantage of instruction-level parallelism. Machine parallelism is determined by the number of instructions that can be fetched, decoded and executed at the same time.

Both instruction level and machine parallelism are important factors in enhancing performance. The use of simple and fixed-length instruction set architecture, RISC, enhances instruction level parallelism.

3.9.2 Instruction-Issue Policy

Considering the limitations of parallelism processor must able to identify the next instruction to be fetched, decoded and executed. This process is known as *instruction issue* and each processor has its own *instruction issue policy*.

Superscalar processors use following instruction issue policies.

- In-order issue with in-order completion
- In-order issue with out-of-order completion
- Out-of-order issue with out-of-order completion

In-order Issue with In-order Completion

In this policy instructions are executed in exact order that would be achieved by sequential execution.

Let us consider the instructions with following constraints.

- I1 requires two cycles to execute
- I3 and I4 conflict for function unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for function unit

With above constraints in-order issue with in-order completion policy executes all instructions as shown in the Fig. 3.9.1.

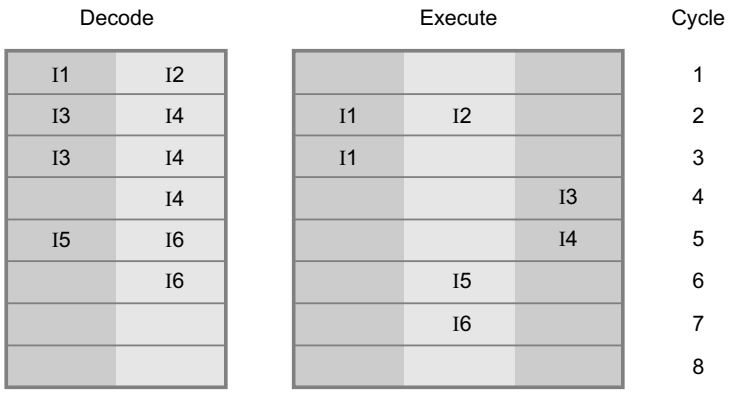


Fig. 3.9.1 In order issue and in order completion

In-order Issue with Out-of-order Completion

If any instruction requires multiple cycles and the next instruction is independent of current instruction then it is allowed to execute before the completion of current instruction. This policy is referred to as in-order issue with out-of-order completion. This policy is used in scalar RISC processors to improve the performance of the instructions that require multiple cycles. Fig. 3.9.2 shows the modified execution sequence due to this policy. As shown in the Fig. 3.9.2 the net result is saving of one cycle.

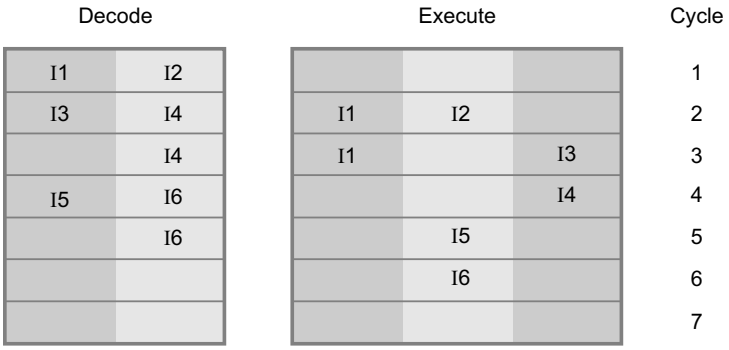


Fig. 3.9.2 In-order issue and out order completion

Out-of-order Issue with Out-of-order Completion

With in-order issue, the processor decodes the instructions upto the point of a dependency or conflict. But in out-of-order issue, processor decodes the instruction beyond any dependency or conflict, to achieve look ahead capacity, allowing it to identify independent instructions that can be brought into the execute stage. To implement this, processor provides buffer commonly known as *instruction window* where decoded instructions are stored. Fig. 3.9.3 shows the modified execution sequence due to this policy

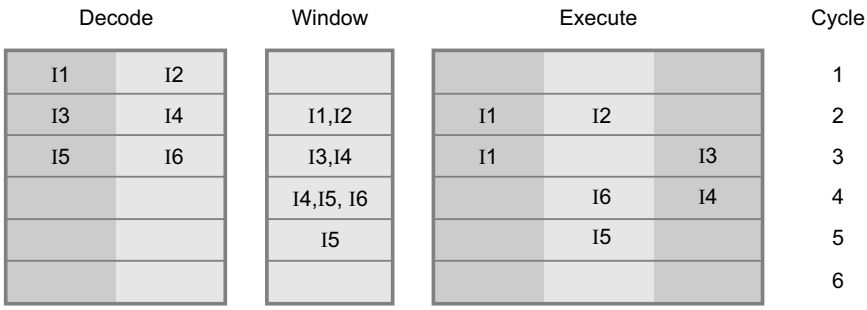


Fig. 3.9.3 Out-of-order issue with out-of-order completion

3.9.3 Register Renaming

The storage conflict can be avoided by duplicating the contents of registers. In this context, the technique is referred to as *register renaming*. To implement register renaming, processor dynamically allocates registers.

3.9.4 Branch Prediction

The branch prediction techniques discussed in section 3.8.8 are used in the supescaler processors to enhance the performance.

Review Questions

1. *What do you mean by superscaler processor ?*
2. *List the issues which must be considered in implementation of superscaler processor.*
3. *Describe the characteristics of supper scalar processing.* **Dec.-11, 12, Marks 8**

3.10 Nano Programming

May-11, Dec.-11

In most microprogrammed processors, an instruction fetched from memory is interpreted by a microprogram stored in a single control memory CM. However, in few microprogrammed processors, the microinstruction do not directly used by the decoder to generate control signals. Instead they are used to access second control memory called a **nanocontrol memory** (nCM). So there are two levels of control memories, a higher-level one known as microcontrol memory μCM and lower level one known as nCM. The μCM stores the microinstructions whereas nCM stores nanoinstructions. Here, decoder uses nanoinstructions from μCM to generate control signals.

Advantages of Nanoprogramming :

- **Reduces total size of required control memory :**

In two level control design technique, the total control memory size S_2 can be given as

$$S_2 = H_m \times W_m + H_n \times W_n$$

where H_m represents number of words in the high level memory
W_m represents the size of word in the high level memory
H_n represents the number of words in the low level memory
W_n represents the size of word in the low level memory

Usually, the microprograms are vertically organised so H_m is large and W_m is small. Nanoprograms, on the other hand, usually have a highly parallel horizontal organisation, which makes W_n large and H_n small. This gives the compatible size for single level control unit as $S_1 = H_m \times W_n$ which is quite larger than S_2.

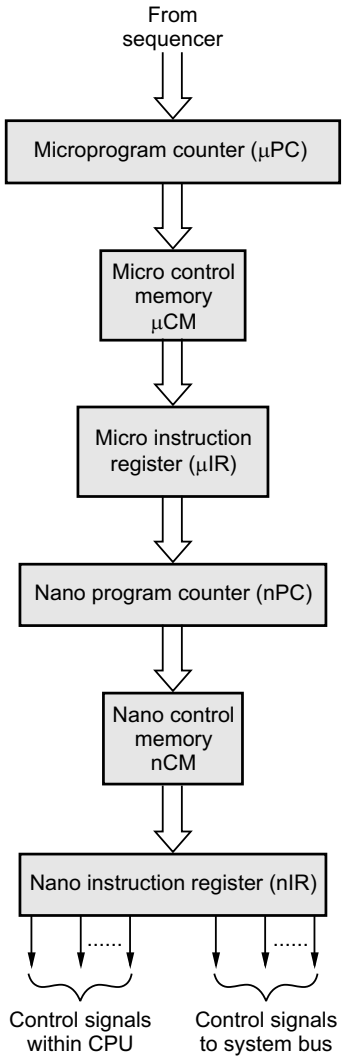


Fig. 3.10.1 Nanoinstruction sequencing organisation

The reduced size of control memory reduces the total chip area.

- **Greater design flexibility :**

Because of two level memory organisation more design flexibility exists between instructions and hardware.

Disadvantage of nano-programming

The main disadvantage of the two-level memory approach is the loss of speed due to the extra memory access required for nano control memory.

Review Question

1. *Write a short note on nano-programming.*
2. *What is meant by nanoprogramming ? Explain the two-level control store organization for nanoprogramming with the help of block diagram.* **May-11, Marks 8**
3. *Write short notes on nano programming.* **Dec.-11, Marks 8**

Two Marks Questions and Answers

Q.1 ***What is instruction cycle ?***

Ans. : The sequence of operations involved in processing an instruction constitutes an instruction cycle.

Q.2 ***What are the major phases of instruction cycle ?***

Ans. : The instruction cycle is subdivided into three major phases : Fetch cycle, decode cycle and execute cycle.

Q.3 ***What are micro-operations ?***

Ans. : To perform fetch, decode and execute cycles the processor unit has to perform set of operations called micro-operations.

Q.4 ***What is data path ?***

Ans. : The data registers, ALU and the interconnecting bus is referred to as **data path**.

Q.5 *What is meant by hardwired control ?*

Ans. : It is the one that contains control units that use fixed logic circuits to interpret instructions and generate control signals from them. Here, the fixed logic circuit block includes combinational circuit that generates the required control outputs for decoding and encoding functions.

Q.6 ***State the advantages of hardwired control unit.***

Ans. : The advantages of hardwired control unit are :

- Hardwired control unit is fast because control signals are generated by combinational circuits.
- The delay generation of control signals depends upon the number of gates.
- It has greater chip area efficiency since its uses less area on-chip.

Q.7 ***State the disadvantages of hardwired control unit.***

Ans. : The disadvantages of hardwired control unit are :

- More is the control signals required by CPU; more complex will be the design of control unit.

- Modifications in control signal are very difficult. That means it requires rearranging of wires in the hardware circuit.
- It is difficult to correct mistake in original design or adding new feature in existing design of control unit.

Q.8 *State the various design methods of hardwired control unit.*

Ans. : The various design methods of hardwired control unit are :

- State-table method
- Delay-element method
- Sequence-counter method
- PLA method.

Q.9 *What is microprogramming ?*

Ans. : Microprogramming is a method of control unit design in which the control signal selection and sequencing information is stored in a ROM or RAM called a **control memory** CM.

Q.10 *What is control memory ?*

Ans. : A memory that is part of a control unit is referred to as a **control memory**. It stores the sequences of micro-operations to be performed to execute microinstructions.

Q.11 *What is microprogram ?*

Ans. : A sequence of one or more microoperations designed to control specific operation, such as addition, multiplication is called a **microprogram**.

Q.12 *What is control word ?*

Ans. : Grouping the control variables at any given time form a string of 1's and 0's, called a **control word**.

Q.13 *What is microinstruction ?*

Ans. : Each word in the control memory is a microinstruction which specifies the control signals to be activated to perform one or more micro-operations.

Q.14 *What is microcode ?*

Ans. : The translation of symbolic microprogram to binary produces a binary microprogram called microcode.

Q.15 *What is the purpose of microprogram sequencer ?*

Ans. : The microprograms for all instructions are stored in the control memory. The address where these microinstructions are stored in CM is generated by microprogram sequencer/microprogram controller.

Q.16 ***State the advantages of microprogrammed control.***

Ans. : The advantages of microprogrammed control are :

- It simplifies the design of control unit. Thus it is both, cheaper and less error prone to implement.
- Control functions are implemented in software rather than hardware.
- The design process is orderly and systematic.
- More flexible, can be changed to accommodate new system specifications or to correct the design errors quickly and cheaply.
- Complex function such as floating point arithmetic can be realised efficiently.
- The new or modified instruction set of CPU can be easily implemented by simply rewriting or modifying the contents of control memory.
- The fault can be easily diagnosed in the micro-program control unit using diagnostics tools by maintaining the contents of flags, registers and counters.

Q.17 ***State the disadvantages of microprogrammed control.***

Ans. : The disadvantages of microprogrammed control are :

- A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.
- The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.
- The design duration of microprogram control unit is more than hardwired control unit for smaller CPU.

Q.18 ***What is the purpose of grouping of control signals ?***

Ans. : Grouping technique is used to reduce the number of bits in the microinstruction.

Q.19 ***Which techniques are used for grouping of control signals ?***

Ans. : The grouping of control signals can be done either by using technique called vertical organisation or by using technique called horizontal organisation.

Q.20 ***Give the comparison between horizontal and vertical organisation.***

Ans. : Refer section 3.5.

Q.21 ***State the advantages and disadvantages of horizontal and vertical organisations.***

Ans. : Refer section 3.4.1.2.

Q,22 ***List out various branching technique used in microprogram control unit ?***

Ans. : a) Bit-oring

b) Using conditional variable

c) Wide branch addressing.

Q.23 ***What is straight-line sequencing ?***

Ans. : Process of fetching and executing an instruction one at a time in order of increasing address with the help of information in program counter is called straight-line sequencing.

Q.24 ***What are the address-sequencing capabilities required in a control memory ?***

Dec.-06

Ans. : Each microinstruction should explicitly or implicitly specify the next microinstruction to be used. Such address sequencing capabilities are required in a control memory.

Q.25 ***Explain MDR and MAR.***

Ans. : The data and address lines of the external memory bus connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus.

Q.26 ***Name two special purpose registers.***

Ans. : a) Index register b) Stack pointer

Q.27 ***Describe the two control signals used for register transfer.***

Ans. : For each register, two control signals are used to place the contents of that register on the bus or to load data on the bus into the register. The input and output of register Ri are connected to the bus via switches controlled by the signals Riin and Riout, respectively. When Riin is set to 1, the data on the bus are loaded into Ri. Similarly, when Riout is set to 1, the contents of register Ri are placed on the bus. While Riout is equal to 0, the bus can be used for transferring data from other registers.

Q.28 ***Define processor clock.***

Ans. : Processor clock is defined as the time periods in which all operations and data transfers with in the processor take place.

Q.29 *What is known as multiphase clocking ?*

Ans. : When edge-triggered flip flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as multiphase clocking.

Q.30 *Define MFC.*

Ans. : To accommodate the variability in response time, the processor waits until it receives an indication that the requested read operation has been completed. The control signal used for this purpose is known as Memory-Function-Completed (MFC).

Q.31 *What is WMFC ?*

Ans. : WMFC is the control signal that causes the processor's control circuitry to wait for the arrival of the MFC signal.

Q.32 *What is mean by branch instruction ?*

Ans. : A branch instruction is an instruction which replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. The location following a branch instruction is called a branch delay slot.

Q.33 *Define register file.*

Ans. : All general purpose registers are combined into a single block called the register file.

Q.34 *What are the two approaches used for generating the control signals in proper sequence ?*

Ans. : 1. Hardwired control

2. Microprogrammed control.

Q.35 *What are the factors determine the control signals?*

Ans. : 1. Contents of the control step counter

2. Contents of the instruction register

3. Contents of the condition code flags

4. External input signals, such as MFC and interrupt requests.

Q.36 *What is control store ?*

Ans. : The microroutines for all instructions in the instruction set of a computer are stored in a special memory called the control store.

Q.37 *What is the drawback of assigning one bit position to each control signal ?* **Dec.-12**

Ans. : Assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large. Moreover, only a few bits are set to '1' in any given microinstruction, which means the available bit space is poorly used.

Q.38 *Name some register output control signals.*

Ans. : Pcout, MDRout, Zout, Offsetout, R0out, R1out, R2out, R3out, and TEMPout.

Q.39 *What is said to be side effect ?*

Ans. : When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect.

Q.40 *What is dispatch unit ?*

Ans. : A separate unit which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. The dispatch unit also performs the decoding function.

Q.41 *Define pipelining.* **May-09, 12, Dec.-10**

Ans. : Pipelining is a technique of decomposing a sequential process into sub operations with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

Q.42 *What is instruction pipelining ?*

Ans. : Performing fetch, decode and execute cycles for several instructions simultaneously to reduce overall processing time is referred to as **instruction pipelining**.

Q.43 *List the four stages in the instruction pipelining.*

Ans. : The four stages in the instruction pipelining are :

S_1 - Fetch (F) : Read instruction from the memory.

S_2 - Decode (D) : Decode the opcode and fetch source operand (s) if necessary.

S_3 - Execute (E) : Perform the operation specified by the instruction.

S_4 - Store (S) : Store the result in the destination.

Q.44 *What is the ideal speed-up expected in a pipelined architecture with 'n' stages ? Justify your answer.* **May-07**

Ans. : Refer example 3.8.5.

Q.45 *What is hazard ?*

Ans. : Any reason that causes the pipeline to stall is called a **hazard**.

Q.46 *List the different types of hazards.*

Ans. : The different types of instruction hazards are :

1. Structural hazards.
2. Data or Data dependent hazards.
3. Instruction or Control hazards.

Q.47 *What is structural hazard ?* **Dec.-08**

Ans. : The hazard that exist because of conflicts due to insufficient resources when even with all possible combination, it may not be possible to overlap the operation is called structural hazard.

Q.48 *What is data hazard in pipelining ? What are the solutions ?* **Dec.-07, 08**

Ans. : When either the source or the destination operands of an instruction are not available at the time expected in the pipeline and as a result pipeline is stalled, we say such a situation is a **data hazard**.

1. The easiest way to handle data hazards is to stall the pipeline.
2. The second simple hardware technique which can handle data hazard is called **forwarding** or **register by passing**.

Q.49 *What are the classification of data hazards ?*

Ans. : The data hazard can be classified as,

1. RAW (read after write) hazard
2. WAW (write after write) hazard
3. WAR (write after read) hazard

Q.50 *What is instruction or control hazard ?*

Ans. : The hazard due to pipelining branch and other instructions that change the contents of program counter is called instruction or control hazard.

Q.51 *What is pipeline scheduling?*

Ans. : Rather than allowing the pipeline to stall, the compiler can rearrange instructions to avoid data hazard. It is called **pipeline scheduling**.

Q.52 *What is delayed load and delayed slot ?*

Ans. : A load which requires that the following instruction do not use its result is said to be **delayed load** and the pipeline slot after load instruction is called **delayed slot**.

Q.53 *How addressing modes affect the instruction pipelining ?*

Ans. : Degradation of performance is an instruction pipeline may be due to address dependency where operand address cannot be calculated without available information

needed by addressing mode for e.g. an instructions with register indirect mode cannot proceed to fetch the operand if the previous instructions is loading the address into the register. Hence operand access is delayed degrading the performance of pipeline.

Q.54 *How Compiler is used in Pipelining ?*

Ans. : A compiler translates a high level language program into a sequence of machine instructions. The number of cycles required to execute program is dependent not only on the choice of instruction, but also on the order in which they appear in the program. The compiler may rearrange program instruction to achieve better performance of course, such changes must not affect of the result of the computation.

Q.55 *What is loop buffer ?*

Ans. : A loop buffer is a small very high speed memory. It is used to store recently prefetched instructions in sequence. If conditional branch is valid, the hardware first checks whether the branch target is within the loop buffer. If so, the next instructions are fetched from the buffer, instead of memory avoiding memory access.

Q.56 *List commonly used branch prediction techniques.*

Ans. : The commonly used branch prediction techniques are :

- Predict Never Taken
- Predict Always Taken
- Predict By Opcode
- Taken/Not Taken Switch
- Branch History Table

Q.57 *What is the use of condition code register ?*

Ans. : The conditional branch instruction checks the condition code flags to decide the flow of program execution.

Q.58 *What is superscaler and superscaler processor ?* **May-12**

Ans. : Processors reach performance levels greater than one instruction per cycle by fetching, decoding and executing several instructions concurrently. This mode of operation is called **superscaler**. A processor capable of parallel instruction execution and having performance level greater than one instruction per cycle is known as **superscaler processor**.

Q.59 *What is instruction issue degree ?*

Ans. : If the processor can issue up to k instructions simultaneously to the various E-units, then k is called instruction issue degree.

Q.60 *Define parallel processing.*

Ans. : Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

Q.61 *What is Register Renaming ?*

Ans. : If a temporary register assumes the role of the permanent register whose data it is holding and is given the same name is called the Register Renaming.

Q.62 *What do you mean by out-of order execution ?*

Ans. : The dispatch unit dispatches the instructions in the order in which they appear in the program. But their execution may be completed in the different order because of data dependency among the instructions. Such situation is called out-of order execution.

Q.63 *What is the function of commitment unit ?*

Ans. : It is a special control unit needed to guarantee in-order commitment when out-of-order execution is allowed. It uses a special queue called the 'reorder buffer' which stores the instruction (s) committed next.

Q.64 *What is instruction throughput ?*

Ans. : The instruction throughput is the number of instructions executed per second. For sequential execution, it is given by,

$$P_s = R/S.$$

Where R is a clock rate measured in clocks per second and S is the average number of steps needed to execute one machine instruction.

Q.65 *Name the two types of branch prediction strategies.*

Ans. : The two types of branch prediction strategies are :

- Static branch strategy
- Dynamic branch strategy

Q.66 *What is branch target buffer ?*

Ans. : The buffer in which the recent branch information is stored is called branch target buffer.

Q.67 *Define processor clock.*

Ans. : Processor circuits are controlled by a timing signal called processor clock, the clock defines regular time interval called clock cycle.

Q.68 *Draw the structure of two stage instruction pipe line.* **Dec.-07**

Ans. : The Fig. 1 shows the structure of two stage instruction pipe line.

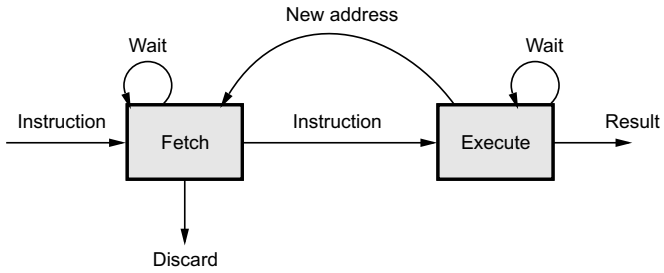


Fig. 1

Q.69 *What is branch folding ?*

Ans. : The instruction fetch unit has executed the branch instruction concurrently with the execution of other instructions. This technique is referred to as branch folding.

Q.70 *What is delayed branching ?*

Ans. : A technique called delayed branching can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions.

Q.71 *Define speculative execution.*

Ans. : Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

Q.72 *What is imprecise and precise exception ?*

Ans. : Situation in which one or more of the succeeding instructions have been executed to completion is called imprecise exception. Situation in which all subsequent instructions that may have been partially executed are discarded. This is called a precise exception.

Q.73 *What is a deadlock ?*

Ans. : A deadlock is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs,

which is being held by unit b. at the same time, unit B is waiting for unit A to finish before it can release that resource.

Q.74 *What is called static and dynamic branch prediction ?* **May-09, Dec.-12**

Ans. : The branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called static branch prediction. Another approach in which the prediction decision may change depending on execution history is called dynamic branch prediction.

Q.75 *What are the address-sequencing capabilities required in a control memory ?*

Dec.-06, CSE/IT

Ans. : Each microinstruction should explicitly or implicitly specify the next microinstruction to be used. Such address sequencing capabilities are required in a control memory.

Q.76 *What factors determine the length of the microinstruction ?* **May-11**

Ans. : The factors which determine the length of the microinstruction are :

- The maximum number of simultaneous micro-operations that must be specified, i.e., the degree of parallelism required at the micro-operation level.
- The way in which the control information is encoded.
- The way in which the address of the next instruction is specified.

Q.77 *State the three part of microinstruction format.*

Ans. : The three parts of microinstruction format are :

- **Condition select field :** It specifies the external condition to be tested in the case of conditional branch instructions.
- **Address field :** It gives the address to be used when a branch condition is satisfied.
- **Control field :** It specifies the control signals that are activated to perform the desired micro-operation.

Q.78 *State various issues that PCU of superscalar processor has to handle to detect and exploit instruction-level parallelism.*

Ans. : Refer section 3.9.

Q.79 *What is the use of microassembler ?* **Dec.-11**

Ans. : A microassembler is used to translate microprograms into executable programs that can be stored in the control memory.

Q.80 *What is microassembly language ?*

Ans. : Symbolic language similar to assembly language is used to write microprograms is called microassembly language.

❑❑❑

4 Memory Organization

Syllabus

Random access memories, Serial-access memories, RAM interfaces, Magnetic surface recording, Optical memories, Multilevel memories, Cache and virtual memory, Memory allocation, Associative memory.

Contents

4.1	Basic Concepts	**Dec.-03, May-08, 09, 12**	Marks 8
4.2	Memory Hierarchy/Multilevel Memory Systems	**Dec.-03, 06, 07, 10**	Marks 16
4.3	Random Access Memories	**May-07, 08, 09, 10, 12, Dec.-07, 10, 12**	Marks 16
4.4	RAM Interfaces	**May-07**	Marks 2
4.5	Advanced DRAMs	**May-09, Dec.-08, 10**	Marks 8
4.6	Serial Access Memories	**May-07, 09, 10, Dec.-07, 08, 10, 11**	Marks 16
4.7	Cache Memory	**May-06, 07, 08, 09, 12, Dec.-03, 06, 07, 08, 11**	Marks 16
4.8	Performance Issues	**May-06, 07, 10**	Marks 6
4.9	Virtual Memory	**May-06, 07, 08, 09, 10, 11 Dec.-06, 07, 08, 10, 12**	Marks 16
4.10	Memory Allocation	**May-11**	Marks 8
4.11	Associative Memory	**May-07, 11, Dec.-03, 12**	Marks 16
4.12	Memory Interleaving	**Dec.-06, 07**	Marks 10

Two Marks Questions with Answers

4.1 Basic Concepts

Dec.-03, May-08, 09, 12

Programs and the data that processor operate are held in the main memory of the computer during execution and the data with high storage requirement is stored in the secondary memories such as floppy disk, hard disk etc. In this chapter, we discuss how this vital part of the computer operates.

Characteristics of Memory Systems

The Table 4.1.1 lists the key characteristics of memory systems.

Location	:	CPU
		Internal (main)
		External (Secondary)
Capacity	:	Word size
		Number of words
Unit of transfer	:	Word
		Block
Access method	:	Sequential access
		Direct access
		Random access
		Associative access
Performance	:	Access time, Cycle time, Transfer rate
Physical type	:	Semiconductor
		Magnetic surface
Physical characteristics	:	Volatile / non-volatile
Organization	:	Erasable / non-erasable

Table 4.1.1

Location : The computer memory is placed in three different locations :

CPU It is in the form of CPU registers and its internal cache memory (16 kbytes in case of pentium)

Internal It is the main memory of the system which CPU can access directly.

External It is in the form of secondary storage devices such as magnetic disk, tapes etc. The CPU accesses this memory with the help of I/O controllers.

Capacity : It is expressed using two terms : Word size and number of words.

Word size It is expressed in bytes (8-bit). The common word sizes are 8, 16 and 32 bits.

Number of word This term specifies the number of words available in the particular memory device. For example, if memory capacity is 4 K × 8, then its word size is 8 and number of words are 4 K = 4096.

Unit of transfer : It is the maximum number of bits that can be read or written into the memory at a time. In case of main memory, most of the times it is equal to *word size.* In case of external memory, unit of transfer is not limited to word size, it is often larger than a word and it is referred to as *blocks.*

Access methods : There are two different methods generally used for memory access.

Sequential access Here, memory is organized into units of data, called *records.* If current record is 1, then in order to read record N, it is necessary to read physical records 1 though N–1. A tape drive is an example of sequential access memory.

Random access Here, each addressable location in memory has an unique address. It is possible to access any memory location at random.

Performance : The performance of the memory system is determined using three parameters :

Access time In case of random access memory, it is the time taken by memory to complete read/write operation from the instant that an address is sent to the memory. On the other hand, for non-random access memory, access time is the time it takes to position the read-write mechanism at the desired location.

Memory cycle time This term is used only in concern with random access memory and it is defined as access time plus additional time required before a second access can commence.

Transfer rate It is defined as the rate at which data can be transferred into or out of a memory unit.

Physical type : Two most common physical types used today are *semiconductor memory* and *magnetic surface memory.*

Physical characteristics :

Volatile/Non-volatile If memory can hold data even if power is turned off, it is called as non-volatile memory ; otherwise it is called as volatile memory.

Erasable/Non-erasable The memories in which data is once programmed cannot be erased are called as non-erasable memories. On the other hand, if data in the memory is erasable then memory is called as erasable memory.

The Table 4.1.2 shows the characteristics of some common memory technologies.

Technology	Storage type	Access method	Alterability	Permanence	Typical access time t_A
Semiconductor RAM	Electronic	Random	Read / Write	Volatile	10 ns
Semiconductor ROM	Electronic	Random	Read only	Non-volatile	10 ns
Magnetic (Hard) disk	Magnetic	Semirandom	Read / Write	Non-volatile	50 ns
Optical disk CD-ROM	Optical	Semirandom	Read only	Non-volatile	100 ms
Erasable optical disk	Optical	Semirandom	Read / Write	Non-volatile	100 ms
Magnetic tape	Magnetic	Serial	Read / Write	Non-volatile	1 s Depend on access location

Table 4.1.2 Characteristics of some common memory technologies

The processor of a computer can usually process instructions and data faster than they are fetched from the memory unit. The memory cycle time, then is the bottleneck in the system. One way to avoid this bottleneck is to use a **cache memory**. Cache memory is a small, fast memory that is inserted between the larger, slower main memory and the processor. It usually holds the currently active segments of a program and their data.

In most modern computers, the physical main memory is not as large as the address space spanned by an address issued by the processor. Here, the virtual memory technique is used to extend the apparent size of the physical memory. It uses secondary storage such as disks, to extend the apparent size of the physical memory.

Comparison between Serial and Random Access Memories

The table 4.1.3 shows the comparison between serial and random access memories.

Sr. No.	Serial access memories	Random access memories
1.	Use serial access method.	Use random access method.
2.	Memory is organized into units of data, called **records**. Records are accessed sequentially. If current record is 1, then in order to read record N, it is necessary to read physical records 1 through N–1.	Each storage location in the memory has an unique address and it can be accessed independently of the other locations.

3.	Memory access time is dependent on the position of storage location.	Memory access time is independent of storage location being accessed.
4.	The time required to bring the desired location into correspondence with a read-write head increases effective access time, so serial access tends to be slower than random access.	Memory access time is less.
5.	Cheaper than random access memories.	Random access memories are comparatively costly.
6.	Nonvolatile memories.	May be volatile or nonvolatile depending on physical characteristics.
7.	Magnetic tape is an example of serial access memories.	Semiconductor memories are random access memories.

Table 4.1.3 Comparison between serial and random access memories

Review Questions

1. *Illustrate the characteristic of some common memory technologies.* **Dec.-03, Marks 8**
2. *Describe the functional characteristics that are common to the devices used to build main and secondary computer memories.* **May-08, Marks 6**
3. *What are the types of memory ?* **May-09, Marks 2**
4. *Discuss in detail about the basic memory concepts.* **May-12, Marks 8**

4.2 Memory Hierarchy/Multilevel Memory Systems

Dec.-03, 06, 07, 10

Ideally, computer memory should be fast, large and inexpensive. Unfortunately, it is impossible to meet all the three of these requirements simultaneously. Increased speed and size are achieved at increased cost. Very fast memory system can be achieved if SRAM chips are used. These chips are expensive and for the cost reason it is impracticable to build a large main memory using SRAM chips. The only alternative is to use DRAM chips for large main memories.

Processor fetches the code and data from the main memory to execute the program. The DRAMs which form the main memory are slower devices. So it is necessary to insert wait states in memory read/write cycles. This reduces the speed of execution. The solution for this problem is come out with the fact that most of the computer programs work with only small sections of code and data at a particular time. In the memory system, small section of SRAM is added along with main memory, referred to as **cache memory**. The program which is to be executed is loaded in the main memory, but the part of program (code) and data that work at a particular time is usually accessed from the cache memory. This is accomplished by loading the active part of code and data from main memory to cache memory. The cache controller looks after this swapping between main memory and cache memory with the help of DMA controller. The cache

memory just discussed is called **secondary cache**. Recent processors have the built-in cache memory called **primary cache**.

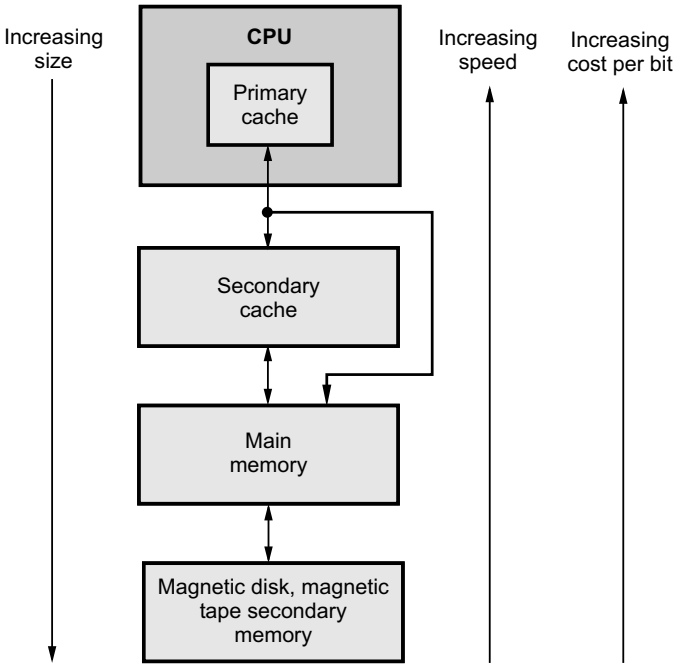


Fig. 4.2.1

DRAMs along with cache allow main memories in the range of tens of megabytes to be implemented at a reasonable cost, the size and better speed performance. But the size of memory is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks and magnetic tapes to implement large memory spaces. Very large disks are available at a reasonable price, sacrificing the speed.

From the above discussion, we can realize that to make efficient computer system it is not possible to rely on a single memory component, but to employ a **memory hierarchy**. Using memory hierarchy all of different types of memory units are employed to give efficient computer system. A typical memory hierarchy is illustrated in Fig. 4.2.1.

In summary, we can say that a huge amount of cost-effective storage can be provided by magnetic disks. A large, yet affordable, main memory can be built with DRAM technology along with the cache memory to achieve better speed performance.

The Fig. 4.2.2 shows how memory hierarchy can be employed in a computer system. As shown in the Fig. 4.2.2, at the bottom of the hierarchy magnetic tapes and magnetic disks are used as a secondary memory. This memory is also known as **auxiliary memory**. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor.

In multiprogramming, a number of independent programs concurrently in process. Multiprogramming, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. Sometimes, it may happen that computer programs are too long to be accommodated in the total space of available in main memory or in different parts of the memory hierarchy. This arises the need for running partial programs for varying the amount of main memory in use by a given program

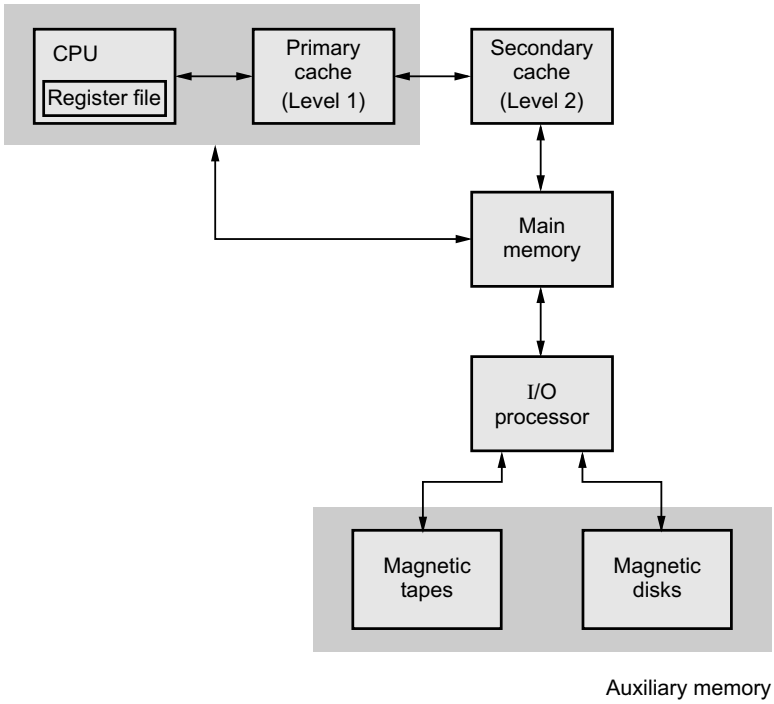


Fig. 4.2.2 Memory hierarchy in a computer system

and for moving programs around the memory hierarchy. We also know that, a computer system uses many programs and all the programs cannot reside in main memory at all times; they normally resides in auxiliary memory. When the program or a part of it is to be executed, it is transferred to main memory. The subsystem in the computer system that controls the flow of information between auxiliary memory and main memory is called the **memory management system.**

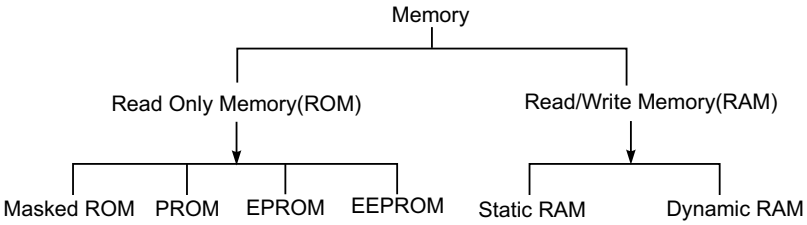
Review Questions

1. *Give the major difference between cache-main and main-secondary memory hierarchies.* **Dec.-03, Marks 2**
2. *Draw a neat sketch of memory hierarchy and explain the need of cache memory.* **Dec.-06, Marks 8**
3. *Explain the concept of memory hierarchy.* **Dec.-07, Marks 8**
4. *Draw the neat sketch of memory hierarchy and explain the need of cache memory.* **Dec.-10, Marks 16**

4.3 Random Access Memories

May-07, 08, 09, 10, 12, Dec.-07, 10, 12

As mentioned earlier, main memory consists of DRAMs supported with SRAM cache. These are semiconductor memories. The semiconductor memories are classified as shown below. In this section, we study various types of semiconductor memories.



Classification of semiconductor memories

4.3.1 ROM (Read Only Memory)

It is a read only memory. We cannot write data in this memory. It is non-volatile memory i.e. it can hold data even if power is turned off. Generally, ROM is used to store the binary codes for the sequence of instructions you want the computer to carry out and data such as look up tables. This is because this type of information does not change.

It is important to note that although we give the name RAM to static and dynamic read/write memory devices, that does not mean that the ROMs that we are using are also not random access devices. In fact, most ROMs are accessed randomly with unique addresses.

There are four types of ROM : Masked ROM, PROM, EPROM and EEPROM or E^2PROM.

4.3.1.1 ROM

We can't write data in read only memories. It is non-volatile memory i.e. it can hold data even if power is turned off. Generally, ROM is used to store the binary codes for the sequence of instructions you want the computer to carry out and data such as look up tables. This is because this type of information does not change.

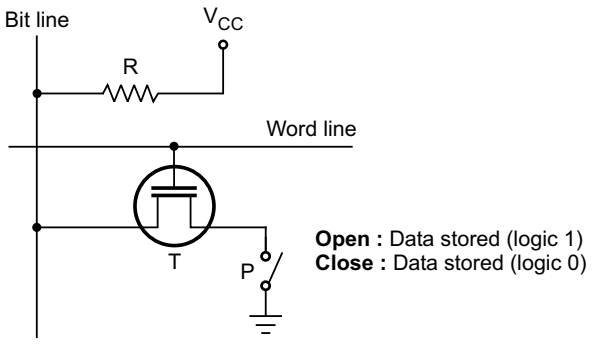


Fig. 4.3.1 ROM cell

It is important to note that although we give the name RAM to static and dynamic read/write memory devices, that does not mean that the ROMs that we are using are

also not random access devices. In fact, most ROMs are accessed randomly with unique addresses.

The Fig. 4.3.1 shows the typical configuration of a ROM cell. It consists of a transistor T and switch P. The transistor T is driven by the word line. The contents of cell can be read from the cell when word line is logic 1. A logic value 0 is read if the transistor is connected to ground through switch P. If switch P is open, a logic value 1 is read. The bit line is connected through a resistor to the power supply. A sense circuit at the end of the bit line generates the proper output value. Data is stored into a ROM when it is manufactured.

There are four types of ROM : Masked ROM, PROM, EPROM and EEPROM or E^2PROM.

4.3.1.2 PROM (Programmable Read Only Memory)

PROMs are programmed by user. To provide the programming facility, each address select and data line intersection has its own fused MOSFET or transistor. When the fuse is intact, the memory cell is configured as a logic 1 and when fuse is blown (open circuit), the memory cell is logical 0. Logical 0s are programmed by selecting the appropriate select line and then driving the vertical data line with a pulse of high current. The Fig. 4.3.2 shows a PROM fused MOSFET memory cell.

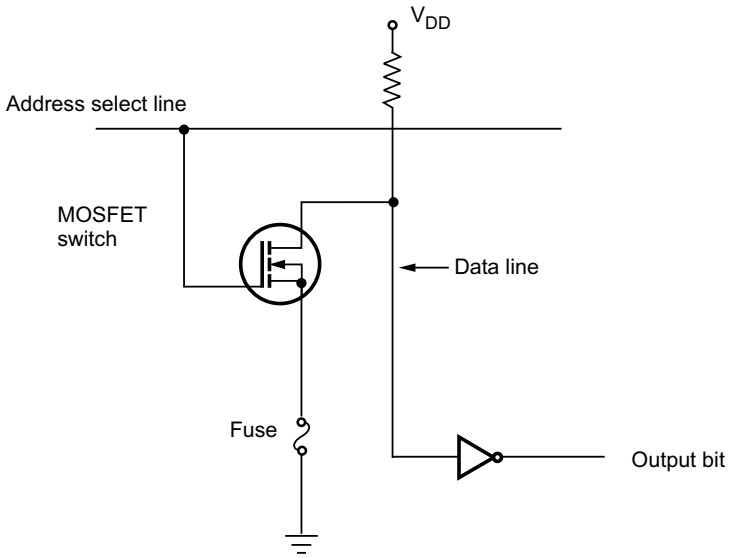


Fig. 4.3.2 Single fused PROM cell

Fig. 4.3.2 shows four byte PROM. It has diodes in every bit position; therefore, the output is initially all 0s. Each diode, however has a fusible link in series with it. By addressing bit and applying proper current pulse at the corresponding output, we can blow out the fuse, storing logic 1 at that bit position. The fuse uses material like nichrome and polycrystalline. For blowing the fuse it is necessary to pass around 20 to 50 mA of current for period 5 to 20 μs. The blowing of fuses according to the truth table is called programming of ROM. The user can program PROMs with special PROM

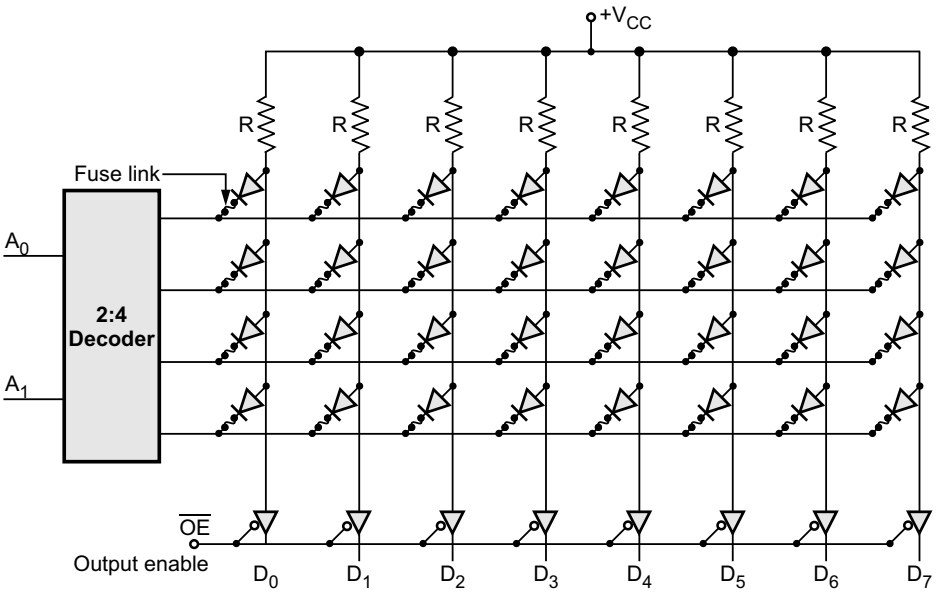


Fig. 4.3.3 Four byte PROM

programmer. The PROM programmer selectively burns the fuses according to the bit pattern to be stored. This process is also known as burning of PROM. The PROMs are one time programmable. Once programmed, the information stored is permanent.

4.3.1.3 EPROM (Erasable Programmable Read Only Memory)

Erasable programmable ROMs use MOS circuitry. They store 1s and 0s as a packet of charge in a buried layer of the IC chip. EPROMs can be programmed by the user with a special EPROM programmer. The important point is that we can erase the stored data in the EPROMs by exposing the chip to ultraviolet light through its quartz window for 15 to 20 minutes, as shown in the Fig. 4.3.4.

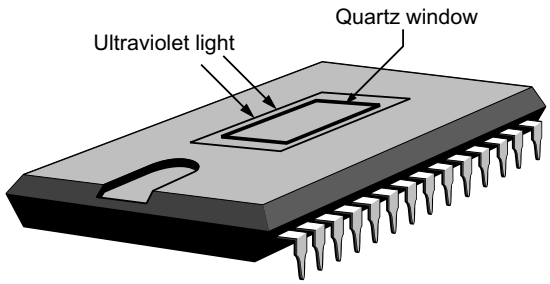


Fig. 4.3.4 EPROM

It is not possible to erase selective information, when erased the entire information is lost. The chip can be reprogrammed. This memory is ideally suitable for product development, experimental projects, and college laboratories, since this chip can be reused many times, over.

EPROM Programming

When erased each cell in the EPROM contains 1. Data is introduced by selectively programming 0s into the desired bit locations. Although only 0s will be programmed, both 1s and 0s can be presented in the data.

During programming address and data are applied to address and data pins of the EPROM. When the address and data are stable, program pulse is applied to the program input of the EPROM. The program pulse duration is around 50 ms and its amplitude depends on EPROM IC. It is typically 5.5 V to 25 V. In EPROM, it is possible to program any location at any time - either individually, sequentially, or at random.

4.3.1.4 EEPROM (Electrically Erasable Programmable Read Only Memory)

Electrically erasable programmable ROMs also use MOS circuitry very similar to that of EPROM. Data is stored as charge or no charge on an insulated layer or an insulated floating gate in the device. The insulating layer is made very thin ($< 200 \mathring{A}$). Therefore, a voltage as low as 20 to 25 V can be used to move charges across the thin barrier in either direction for programming or erasing. EEPROM allows selective erasing at the register level rather than erasing all the information since the information can be changed by using electrical signals. The EEPROM memory also has a special chip erase mode by which entire chip can be erased in 10 ms. This time is quite small as compared to time required to erase EPROM, and it can be erased and reprogrammed with device right in the circuit. However, EEPROMs are most expensive and the least dense ROMs.

4.3.2 RAM (Random Access Memory)

Unlike ROM, we can read from or write into the RAM, so it is often called read/write memory. The numerical and character data that are to be processed by the computer change frequently. These data must be stored in type of memory from which they can be read by the microprocessor, modified through processing and written back for storage. For this reason , they are stored in RAM instead of ROM. But it is a volatile memory, i.e. it cannot hold data when power is turned off.

There are two types of RAMs :

- Static RAM
- Dynamic RAM

4.3.2.1 Static RAM

Memories that consists of circuits capable of retaining their state as long as power is applied are known as **static memories**. These are Random Access Memory (RAM) and hence combinely called **static RAM memories**.

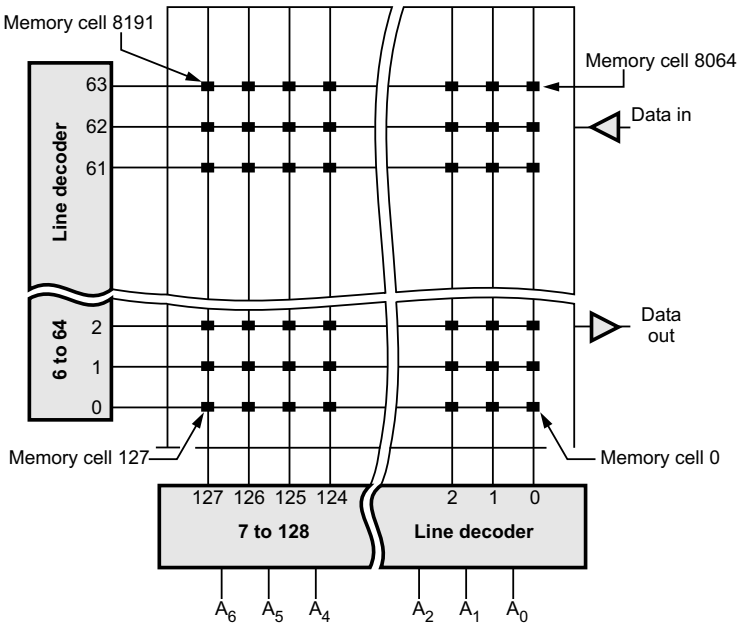


Fig. 4.3.5 Row and column organization for 8192 bit static RAM

Static RAM Cell

The Fig. 4.3.6 shows the implementation of static RAM cell. It consists of two cross-coupled inverters as a latch and two transistors T_1 and T_2 which act as a switches.

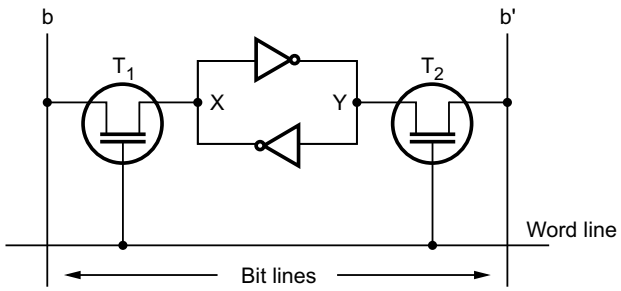


Fig. 4.3.6 Static RAM Cell

The latch is connected to two bit lines by transistors T_1 and T_2. The word line controls the opening and closing of transistors T_1 and T_2. When word line is at logic 0 level (Ground level), the transistors are off and the latch retains its state.

Read Operation

For read operation, word line is made logic 1 (high) so that both transistors are ON. Now if the cell is in state 1, the signal on bit line b is high and the signal on bit line b′ is low. The opposite is true if the cell is in state 0. The b and b′ are complements of each other. The sense/write circuits connected to the bit lines monitor the states of b and b′ and set the output accordingly.

Write Operation

For write operation, the state to be set is placed on the line b and its complement is placed on line b′ and then the word line is activated. This action forces the cell into the corresponding state and write operation is completed.

CMOS Cell

The Fig. 4.3.7 shows the CMOS cell. Here, transistor pairs (T_3, T_5) and (T_4, T_6) form the cross-coupled inverters and transistors T_1 and T_2 act as a switches. The latch is connected to two bit lines by transistors T_1 and T_2.

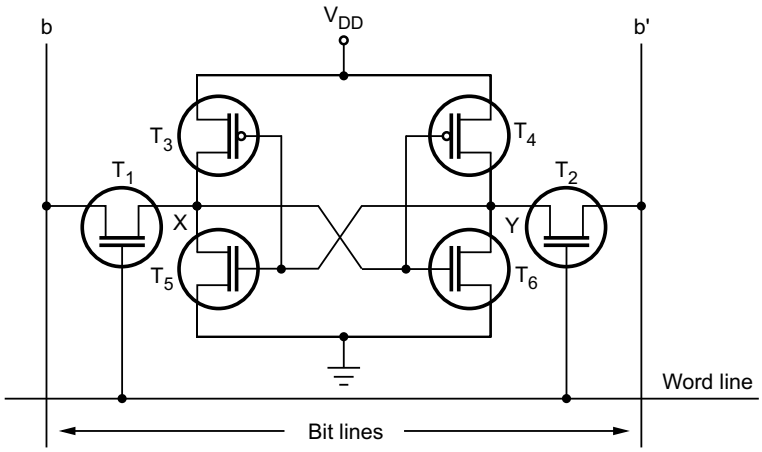


Fig. 4.3.7 CMOS memory cell

The word line controls the opening and closing of transistors T_1 and T_2. When word line is at logic 0 level (ground level), the transistors T_1 and T_2 are off and the latch retains its state.

Read Operation

For read operation, word line is made high to switch-on transistor T_1 and T_2. The cell is in state 1, if voltage at point X is maintained high by having transistors T_3 and T_6 on, while T_4 and T_5 are off. The cell is in state 0, if the voltage at point X is maintained low by having transistors T_3 and T_6 off, while T_4 and T_5 are on.

Write Operation

For write operation, the state to be set is placed on the line b and its complement is placed on line b′, and then the word line is activated. This action forces the cell into the corresponding state.

4.3.2.2 Internal Organization

Memory cells are organized in the form of an array, in which each cell is capable of storing one bit of information. Let us see the organization of memory chip using single decoder and using two dimensional decoding.

Organization using Single Decoder

The Fig. 4.3.8 shows the organization of 16 × 8 (16 words of 8-bit each) memory. Here, the organisation is shown with detail connections of address lines, data lines and control lines. The data input and the data output of each sense/write circuit are connected to a single bidirectional data line that can be connected to the data bus of a computer. Two control lines, R/$\overline{W}$ and $\overline{CS}$, are provided in addition to address and data lines. The R/$\overline{W}$ (Read/$\overline{Write}$) input specifies the required operation and the $\overline{CS}$ input selects a given chip in a multichip memory system.

As shown in the Fig. 4.3.8 there are 16 words of 8-bits each. A memory with 16 words needs four address lines. The four address inputs go through a 4 × 16 decoder to select one of the sixteen words. The decoder is enabled with a memory enable input. When the memory enable is 0, all outputs of the decoder are 0 and none of the memory words are selected. With the memory select at 1, one of the sixteen words is selected, dictated by the value in the four address lines. Once a word has been selected, the read/write input determines the operation. During write operation, the data available in the input lines are transferred into the eight memory cells of the selected word. The memory cells that are not selected are disabled and their previous binary values remain unchanged.

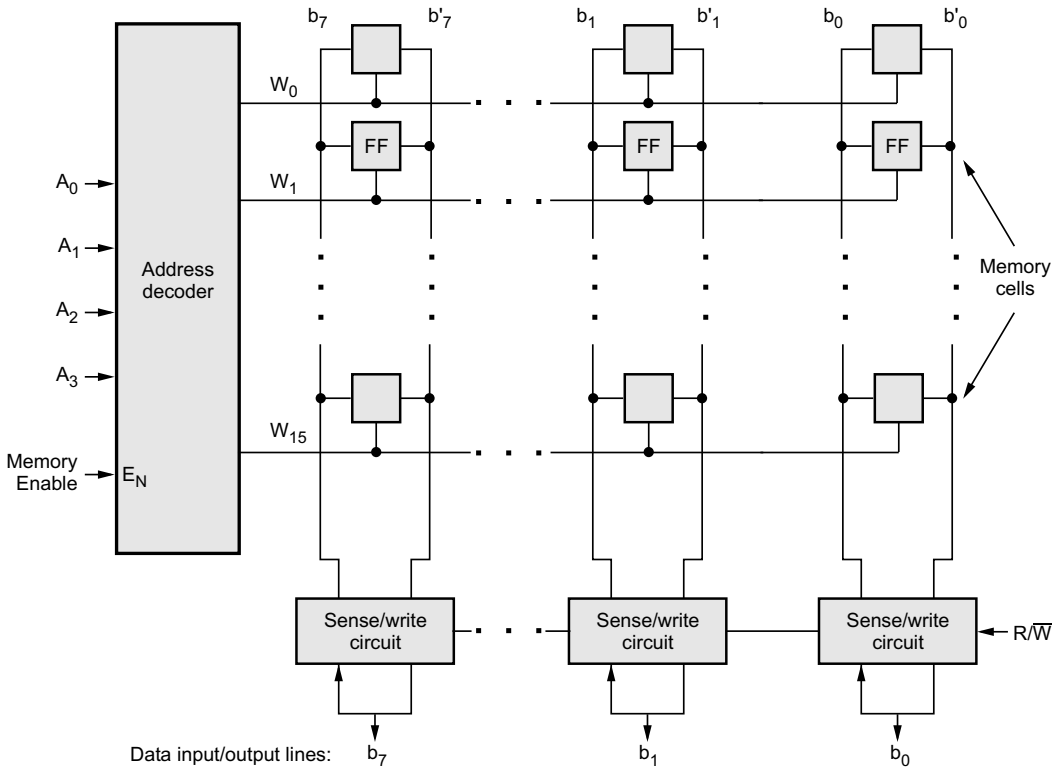


Fig. 4.3.8 Organization of bit cells in a memory chip

Organization using Two Dimensional Decoding

A decoder with k inputs and 2k outputs requires 2k AND gates with k inputs per gate. We can reduce the total number of gates and the number of inputs per gate by employing two decoders in a two-dimensional selection scheme. The Fig. 4.3.9 shows the two dimensional decoding scheme. Here, two k/2 input decoders are used instead of one k-input decoder. One decoder performs the row selection and the other the column selection in a two dimensional matrix configuration.

The two-dimensional selection pattern shown in Fig. 4.3.9 has k = 10, i.e. 10 decoder inputs. These decoder inputs are divided by 2 to get individual decoder inputs, i.e. 5 input for each decoder. Therefore, here instead of using a single 10 × 1024 decoder, we use two 5 × 32 decoders. With a single decoder we would need 1024 AND gates with 10 inputs in each. In the two- decoder case, we need 64 AND gates with five inputs in each. The five most significant bits of the address go to input A and the five least significant bits go to input B. Each word within the memory array is selected by the coincidence of one A and one B line. Thus, each word in memory is selected by the coincidence between 1 of 32 rows and 1 of 32 columns for a total of 1024 words.

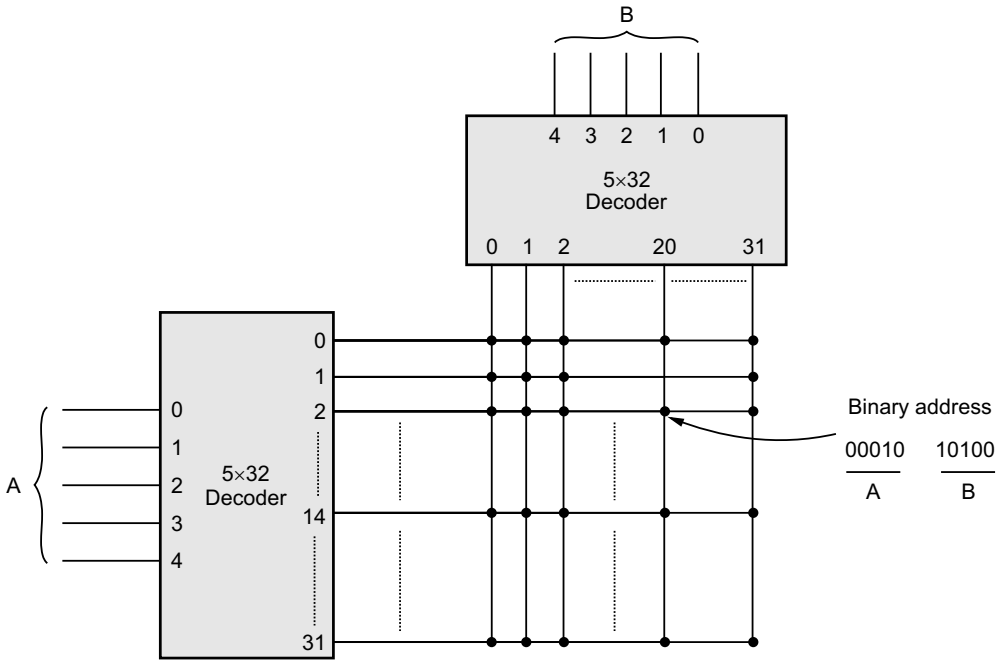


Fig. 4.3.9 Two dimensional decoding scheme

The Fig. 4.3.10 shows the memory organization for 8192 bit static RAM. As shown in the Fig. 4.3.10 the chip has 13 address lines. The first seven address lines are connected to the column decoder to indicate one of the 128 columns. The remaining 6 address lines are connected to the row decoder to indicate one of 64 rows. Where the decoded row

and column cross, they select the desired individual memory cell. Simple arithmetic shows that there are $64 \times 128 = 8192$, crossings. Therefore, this memory has 8192 memory cells.

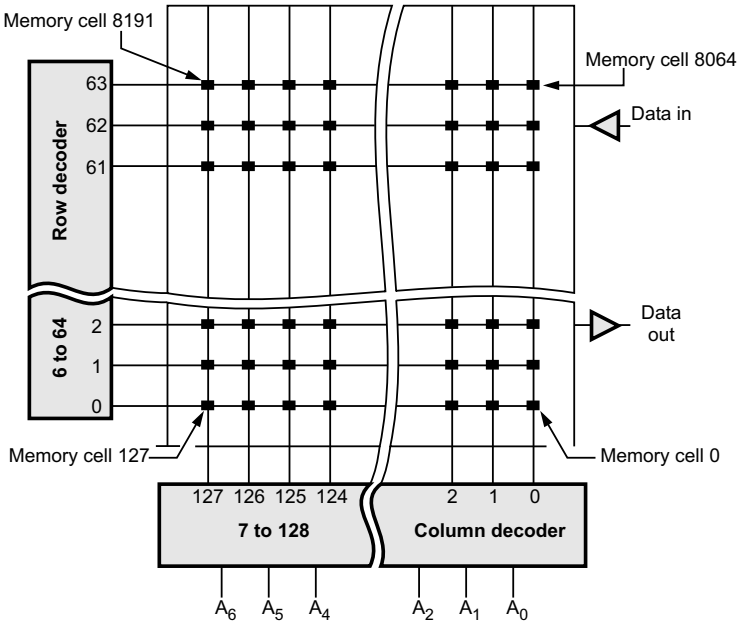


Fig. 4.3.10 Row and column organization for 8192 bit static RAM

4.3.2.3 Dynamic RAM (DRAM)

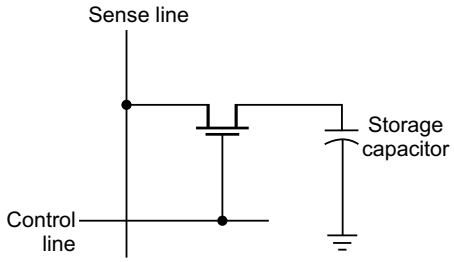


Fig. 4.3.11 Dynamic RAM

Dynamic RAM stores the data as a charge on the capacitor. Fig. 4.3.11 shows the dynamic RAM cell. A dynamic RAM contains thousands of such memory cells. When COLUMN (Sense) and ROW (Control) lines go high, the MOSFET conducts and charges the capacitor. When the COLUMN and ROW lines go low, the MOSFET opens and the capacitor retains its charge. In this way, it stores 1-bit. Since only a single MOSFET and capacitor are needed, the dynamic RAM contains more memory cells as compared to static RAM per unit area.

The disadvantage of dynamic RAM is that it needs refreshing of charge on the capacitor after every few milliseconds. This complicates the system design, since it requires the extra hardware to control refreshing of dynamic RAMs.

4.3.2.4 Internal Structure of DRAM

The Fig. 4.3.12 shows a block diagram of the internal structure of a 64 K× 1 DRAM. In DRAMs, row address and column address lines are multiplexed to reduce number of pins. Here, the logical array has 64 K× 1 bits, but the physical array is square, containing 256 × 256 bits. A complete 16-bit address is presented to the chip in two phases controlled by two signals :

- RAS_L (row address strobe)
- CAS_L (column address strobe)

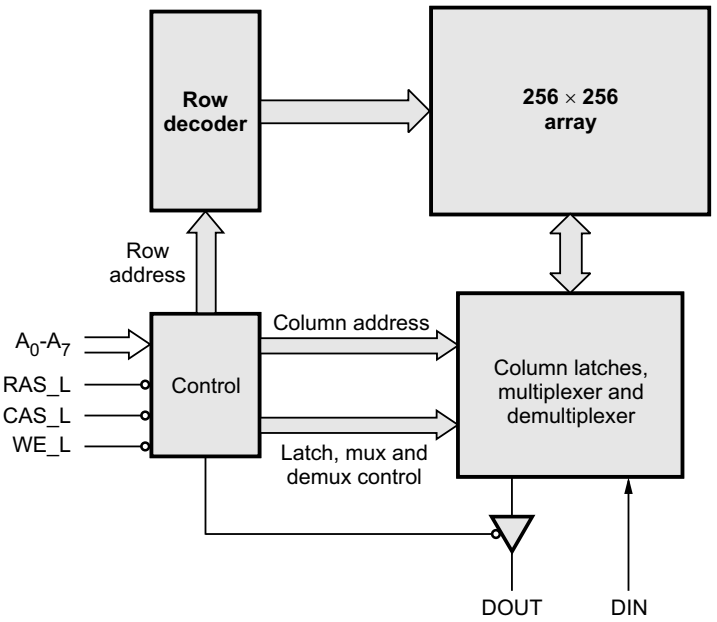


Fig. 4.3.12 Internal structure of 64× 1 DRAM

4.3.3 Comparison between SRAM and DRAM

Sr. No.	Static RAM	Dynamic RAM
1.	Static RAM contains less memory cells per unit area.	Dynamic RAM contains more memory cells as compared to static RAM per unit area.
2.	It has less access time hence faster memories.	Its access time is greater than static RAMs.
3.	Static RAM consists of number of flip-flops. Each flip-flop stores one bit.	Dynamic RAM stores the data as a charge on the capacitor. It consists of MOSFET and the capacitor for each cell.
4.	Refreshing circuitry is not required.	Refreshing circuitry is required to maintain the charge on the capacitors after every few milliseconds. Extra hardware is required to control refreshing. This makes system design complicated.
5.	Cost is more.	Cost is less.

In above discussion we have seen memory organization for ROM and Static RAM. In this organization, data is organized as a single bit data word. However, in most memory ICs the data is organized as eight bit data word. In these ICs eight memory cells are connected in parallel and enabled at a time to read or write eight bit data word. The eight memory cells connected in parallel forms a register and memory is nothing but an array of registers.

Review Questions

1. *Describe the organization of a typical RAM chip.* **May-07, Marks 6**
2. *Write note on static memories.* **Dec.-07, Marks 8**
3. *Write note on ROM technologies.* **Dec.-07, Marks 4**
4. *Give the structure of semiconductor RAM memories. Explain the Read and Write operations in detail.* **Dec.-07, Marks 16**
5. *List the differences between static RAM and dynamic RAM.* **May-08, Marks 2**
6. *Give the features of a ROM cell.* **May-08, Marks 2**
7. *Briefly explain any four nonvolatile memory in detail.* **May-09, Marks 10**
8. *Draw a CMOS memory cell and explain its function.* **May-09, Marks 6**
9. *Explain in detail internal organization of memory chip.* **May-10, 12, Marks 16**
10. *Define random access memory.* **Dec.-10, Marks 2**
11. *Write a note on random access memories.* **Dec.-12, Marks 8**

4.4 RAM Interfaces

May-07

The maximum size of the memory that can be used in any computer is determined from the number of address lines provided by the processor used in the computer. For example, if processor has 20 address lines, it is capable of addressing upto $2^{20} = 1$ M(Mega) memory locations. Similarly, processors whose instructions generate 32-bit address can access a memory that contains upto 2^{32} = 4 G (Giga) memory locations. The number of locations represents the size of the address space of the computer.

We have seen that, most computers are byte addressable and they can have big-endian or little endian arrangement. The Intel processors use little-endian arrangement while big-endian arrangement is used in the Motorola 68000 processor. The ARM architecture can be configured to use either arrangement.

The maximum number of bits that can be transferred from memory or to the memory depend on the data lines supported by the processor. The number of data lines supported by processor gives the word length of the processor and hence the computer.

The control lines from the processor decides the memory operation. In case of read operation, $\overline{RD}$ signal is activated. It is used to enable the active low output enable signal of the memory. In case of write operation, $\overline{WR}$ signal is activated to indicate the write operation. The data transfer between the memory and processor takes place through the use of two processor registers, usually called MAR (Memory Address Register) or simply AR (Address Register) and MDR (Memory Data Register) or simply DR (Data Register). This is illustrated in Fig. 4.4.1. If MAR is k-bit long and MDR is n-bit long, it is possible to across upto 2^k memory locations and during one memory cycle it is possible to transfer n-bit data.

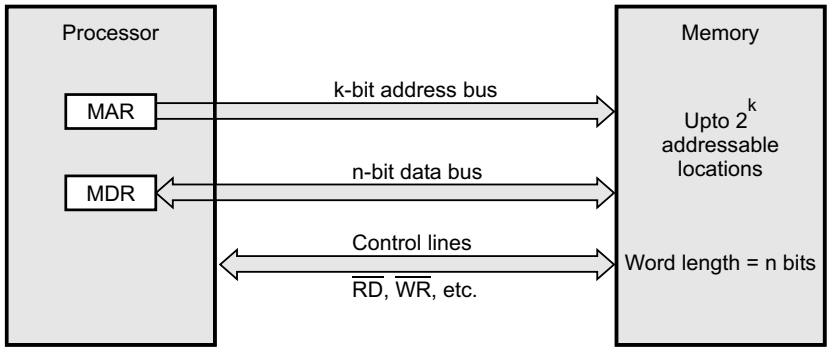
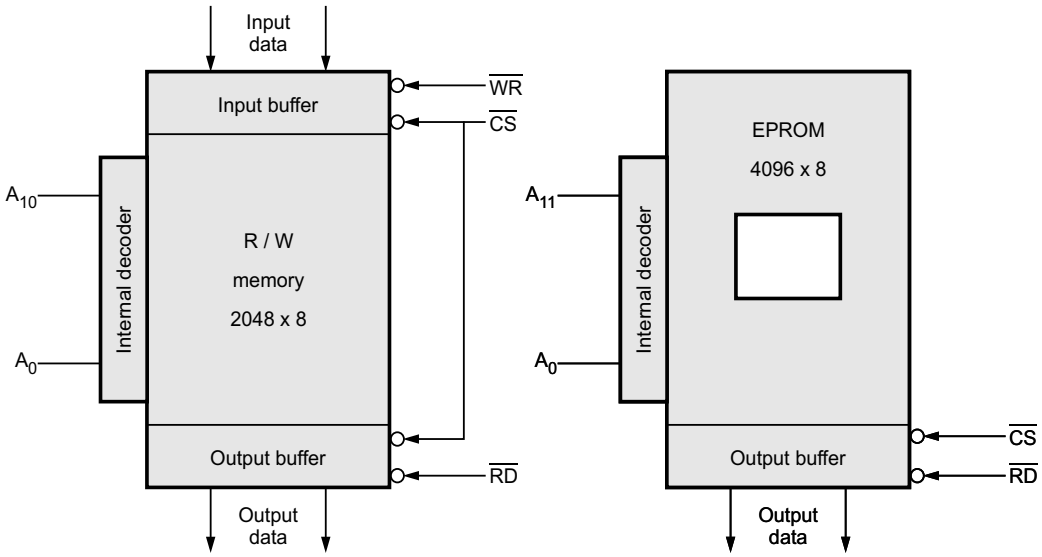


Fig. 4.4.1 Connection between memory and processor

As mentioned earlier, read/write memories consist of an array of registers, in which each register has unique address. The size of the memory is N × M as shown in Fig. 4.4.2 (a) where N is the number of registers and M is the word length, in number of bits.



(a) Logic diagram for RAM **(b) Logic diagram for EPROM**

Fig. 4.4.2

Example 1 : If memory is having 12 address lines and 8 data lines, then

$$\text{Number of registers/memory locations} = 2^N = 2^{12}$$
$$= 4096$$
$$\text{Word length} = \text{M-bit}$$
$$= \text{8-bit}$$

Example 2 : If memory has 8192 memory locations, then it has 13 address lines.

The Table 4.4.1 summarizes the memory capacity and address lines required for memory interfacing.

Memory capacity	Address lines required
1 K = 1024 memory locations	10
2 K = 2048 memory locations	5
4 K = 4096 memory locations	12
8 K = 8192 memory locations	13
16 K = 16384 memory locations	14
32 K = 32768 memory locations	15
64 K = 65536 memory locations	16

Table 4.4.1

As shown in the Fig. 4.4.2 (a) memory chip has 12 address lines A_0-A_{11}, one chip select ($\overline{CS}$) and two control lines, read ($\overline{RD}$) to enable output buffer and write ($\overline{WR}$) to enable the input buffer. The internal decoder is used to decode the address lines. Fig. 4.4.2 (b) shows the logic diagram of a typical EPROM (Erasable Programmable Read-Only Memory) with 4096 (4 K) registers. It has 12 address lines A_0-A_{11}, one chip select ($\overline{CS}$), one read control signal. Since EPROM is a read only memory, it does not require the ($\overline{WR}$) signal.

4.4.1 Expanding Word Size and Memory Capacity

It is possible to expand word size of memory by connecting two or more ICs together. The word size of memory IC can be increased by connecting two memory ICs in such a way that their data bus is in series and address bus in parallel. Both memory ICs are selected simultaneously by common chip select signal to access entire expanded word at a time. The example 4.4.1 illustrates method of word expansion.

Example 4.4.1 *Design 1 K × 8 RAM using two 2114 ICs.*

Solution :

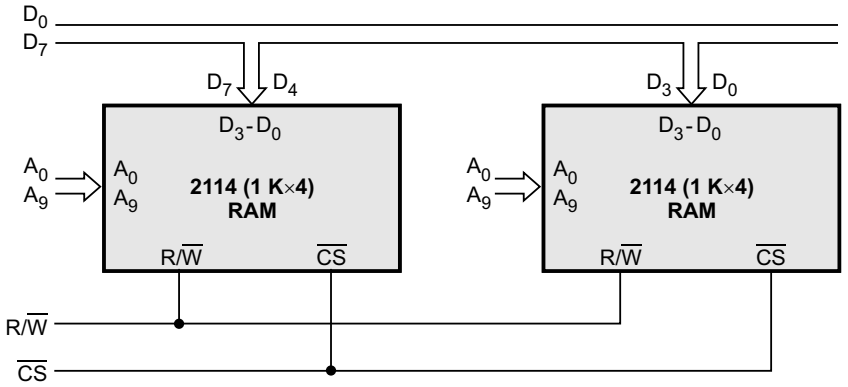


Fig. 4.4.3 Expanding word size

The memory capacity can be increased by connecting two or more memory ICs in parallel. This means that the address, data and control lines are connected in parallel to all memory ICs. Each IC is selected by the separate chip select signal generated by the address decoder. This is illustrated by the following example.

Example 4.4.2 *Design 2 K × 4 RAM using two 2114 ICs.*

Solution :

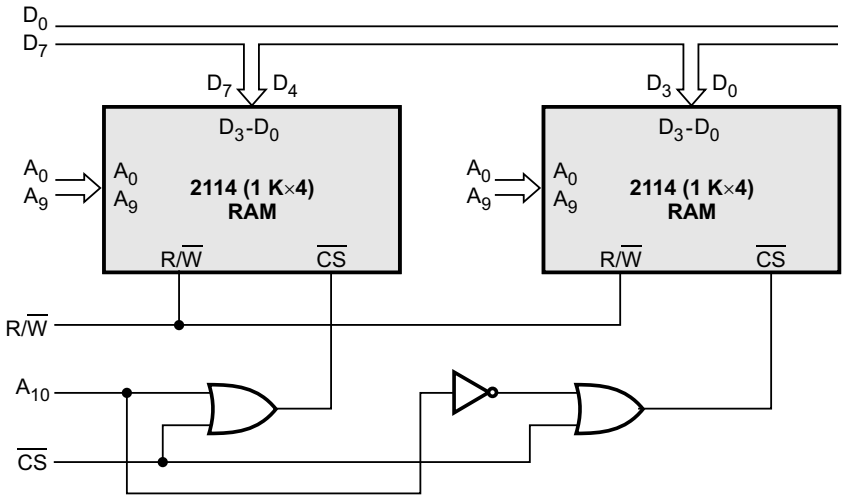


Fig. 4.4.4

The Fig. 4.4.5 shows the implementation of 2 M words of 32-bit each using 512 K× 8 static memory chip. In this implementation, we have expand word size as well as capacity of the memory. Since each memory chip has byte organization, we have to use

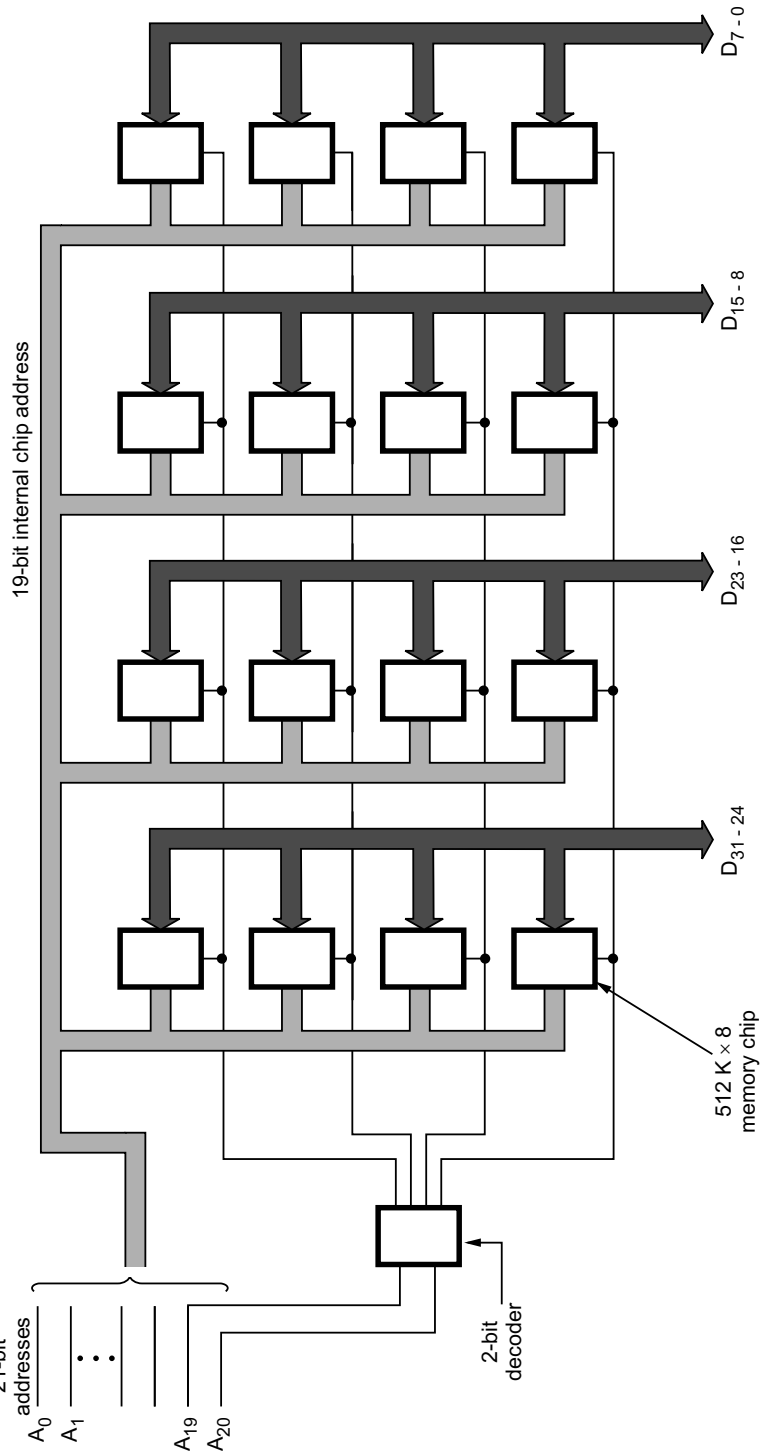

D7 - 0
D15 - 8
D23 - 16
D31 - 24
19-bit internal chip address
512 K × 8 memory chip
21-bit addresses
A0
A1
A19
A20
2-bit decoder
512 K × 8 memory chip
8-bit data input/output
19-bit address
Chip select

Fig. 4.4.5

4 chips in each row to get the word size of 32 bits (8 × 4). Similarly, to increase memory capacity up to 2 M we have to use 4 memory chips of capacity 512 K in each column.

Example 4.4.3 *For a 32 K memory, obtain memory map to interface memory to 16-bit processor.*

Solution :

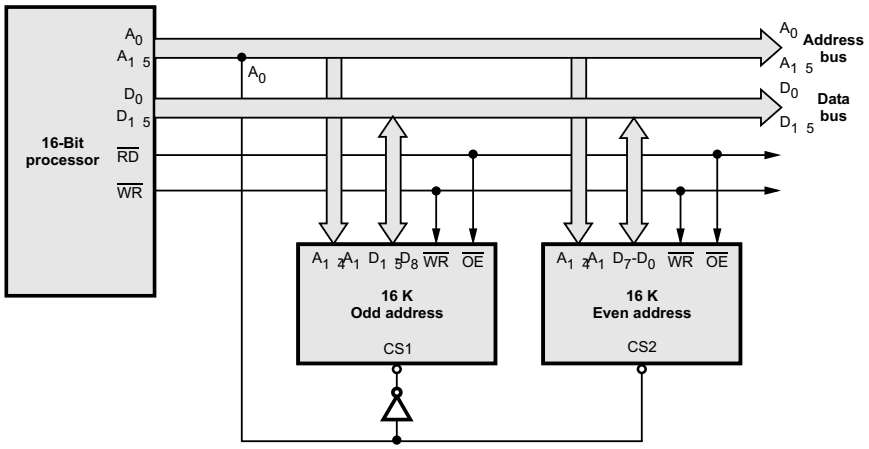


Fig. 4.4.6

Memory Map :

Memory	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
Even address	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H
	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	7FFEH
Odd address	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0001H
	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFFH

Note : A_{15} is considered logic 0.

Example 4.4.4 *How many 128 × 8 memory chips are needed to provide a memory capacity of 4096 × 16 ?*

Solution :

$$\text{Number of chips required} = \frac{\text{Required RAM size}}{\text{Available chip capacity}} = \frac{4096 \times 16}{128 \times 8} = \mathbf{64}$$

Hence, 64 chips of 128 × 8 size are needed to provide a memory capacity of 4096 × 16.

Example 4.4.5 *Given a 32 × 8 size ROM chip with an enable input. Show the external connections necessary to construct a 128 × 8 ROM with four chips and decoder.*

Solution : Address bus width = $\log_2 128 = \log_2 2^7$ = 7-bits = A_6 - A_0

Data bus width = 8-bits = D_7 - D_0

Address lines required for 32 × 8 ROM = $\log_2 3^2 = \log_2 2^5$ = 5 bits = A_4 - A_0

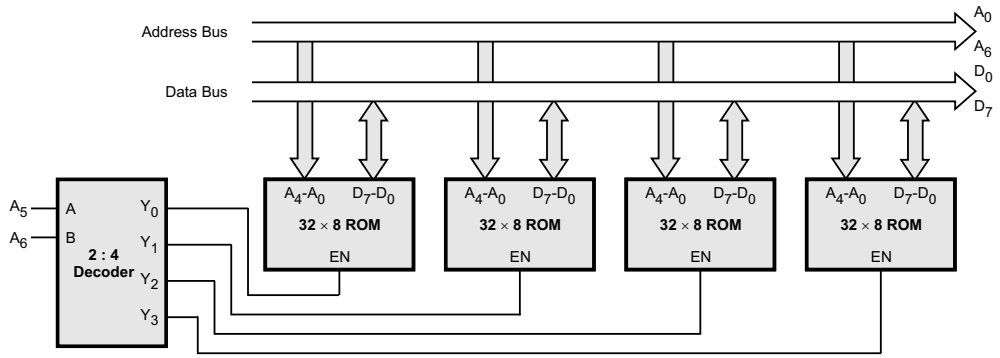


Fig. 4.4.7

Example 4.4.6 *A microprocessor uses RAM chips of 1024 × 1 capacity.*

i) How many chips are needed and how should their address lines be connected to provide a memory capacity of 1024 bytes ?

ii) How many chips are needed to provide a memory capacity of 16 kbytes ? Explain in words how the chips are to be connected to the address bus.

Solution : i) Eight chips are needed with address lines connected in parallel as shown in the Fig. 4.4.8.

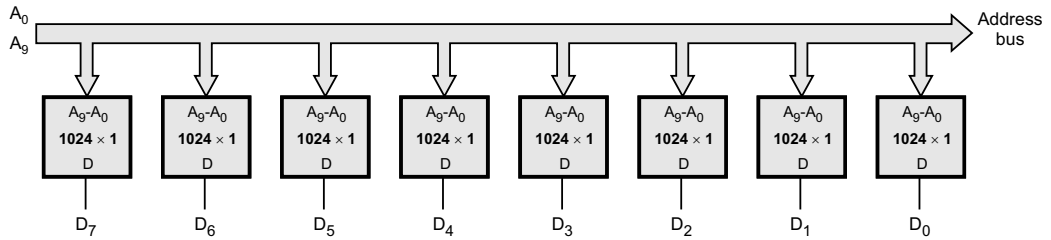


Fig. 4.4.8

ii) $$\text{Number of chips needed} = \frac{\text{Required memory capacity}}{\text{Chip size}}$$

$$= \frac{16\,K \times 8}{1\,K \times 1} = \textbf{128 chips}$$

The address lines A_9 - A_0 from the address bus are connected in parallel to all the chips. In all there are 128 chips. These 128 chips are divided into 16 groups consists of 8 chips in each group. The address lines A_{13} - A_{10} are decoded to generate 16 chip select signals one for each group so that at a time all chips from one group will be selected when corresponding chip select signal is activated.

Review Question

1. *How do you construct a* $8M \times 32$ *memory using* $512k \times 8$ *memory chips ?* **May-07, Marks 2**

4.5 Advanced DRAMs

May-09, Dec.-08, 10

4.5.1 Enhanced DRAM (EDRAM)

Recall DRAM organization. It contains thousands of cells, which are arranged in the matrix form. A particular cell can be accessed by selecting appropriate row and column. In enhanced DRAM, a small SRAM cache is available on the DRAM chip, which is used to store the entire contents of the last row read. This means that, when a particular cell is read from DRAM, the entire contents of row of that cell are stored in the SRAM. Due to this, when data from the same row is accessed, it is read from SRAM instead of DRAM, resulting fast access.

Fig. 4.5.1 shows the organization of 4 Mbit enhanced dynamic RAM. It consists of 2048 bit SRAM. In this, the row address of the last access is latched in the Last-Row-Read Address Latch. A comparator compares the new row address with the last row address to determine whether to read the data from SRAM or DRAM. If comparison is true the data is read from SRAM; otherwise it is read from DRAM and the contents of SRAM are updated with the contents of recent row. (See Fig. 4.5.1 on next page).

Other features of EDRAM :

- The DRAM can be refreshed in parallel with cache read operation. This minimizes the time for which chip is unavailable due to refresh.
- It is also important to note that the read path from the row cache to the output port is independent of the write path from the I/O module to the sense amplifiers. This enables a subsequent read access to the cache to be satisfied in parallel with the completion of the write operation.

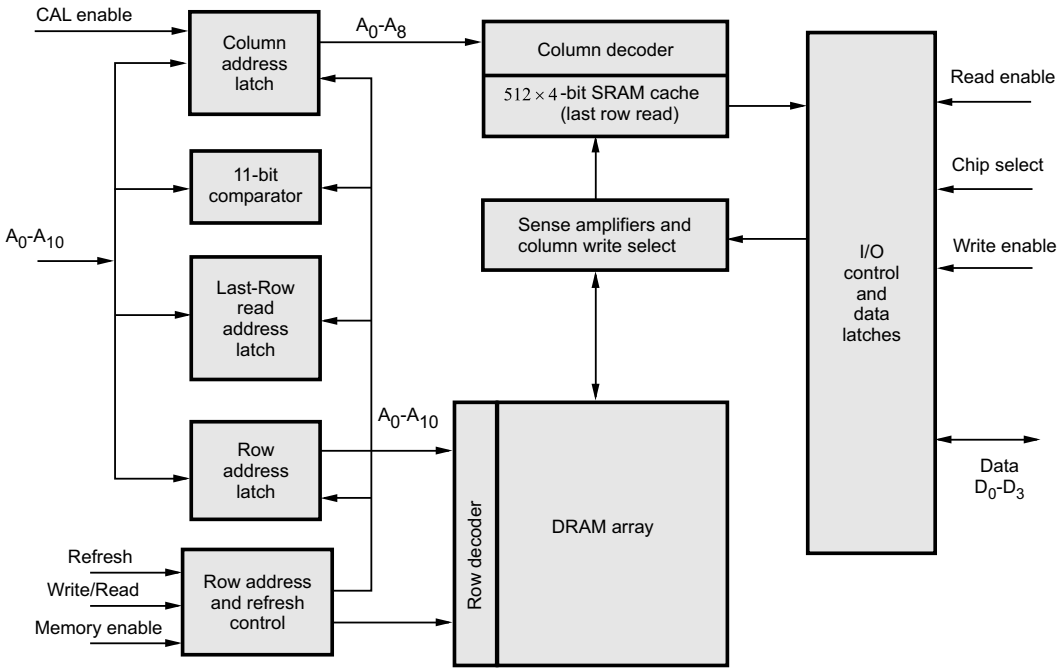


Fig. 4.5.1 Enhanced Dynamic RAM (EDRAM)

4.5.2 Cache DRAM (CDRAM)

The cache DRAM is similar to the EDRAM. The CDRAM contains a larger SRAM cache than the EDRAM (16 kbits Vs 2 kbits). Unlike EDRAM, the SRAM cache of CDRAM is used as a true cache, consisting of a number of 64-bit lines.

The SRAM of the CDRAM can also be used as a buffer to support the serial access of a block of data. For example, to refresh bit mapped video screen, the CDRAM can be used to fetch the data. In such cases, CDRAM prefetches the data from the DRAM into the SRAM buffer and subsequent accesses are made from SRAM, resulting fast accesses.

4.5.3 Synchronous DRAM (SDRAM)

Most recent developments in memory technology have resulted in DRAMs whose operation is directly synchronised with a clock signal. Such memories are known as synchronous DRAMs or simply SDRAMs.

To improve the DRAM performance, SDRAM uses a quite different approach. Let us recall the typical DRAM operation. In a typical DRAM, the processor sends addresses and control signals to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay of access time, the DRAM either writes or reads the data. During this delay the DRAM performs various internal functions, such as selection of row and column lines, sensing

the data, and routing the data in or out. The processor has to wait through this delay slowing the system performance.

To solve above problem, the SDRAM exchanges data with the processor synchronized to an external clock signal. This allows processor to read/write data with memory without imposing wait states. With synchronous access, the DRAM moves data in and out under control of the system clock. In this, DRAM latches the address sent by the processor and then responds after a set number of clock cycles. Meanwhile, the processor can do other task while the SDRAM is processing the request.

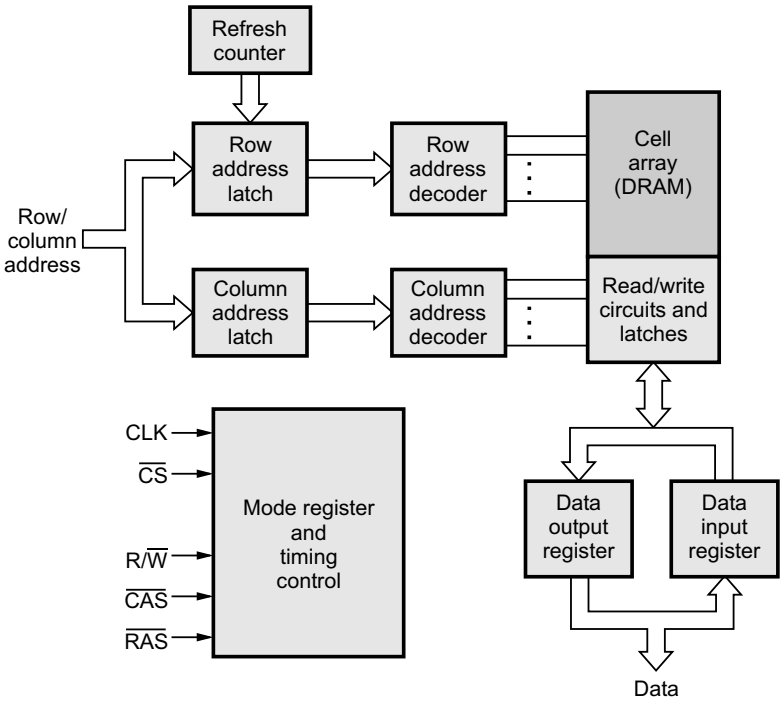


Fig. 4.5.2

The Fig. 4.5.2 shows the internal architecture of SDRAM. The cell organisation of SDRAM is same as asynchronous DRAM. The address and data lines are buffered by means of buffer registers. It supports burst mode transfer.

The SDRAMs can be configured in different modes of operation. These operations can be selected by writing corresponding control information into a mode register. For example, for burst operations we have specify the length. The burst operations we have specify the length. The burst operations use the block transfer capability. The block transfer capability is also known as the **fast page mode** feature. In SDRAMs it is not necessary to provide externally generated pulses on the CAS line to select successive columns. The necessary control signals are provided internally using a column counter and the clock signal.

4.5.3.1 Timing Diagram

The Fig. 4.5.3 shows a timing diagram for a typical burst read cycle of length 4. The first clock cycle row address is latched by activating $\overline{RAS}$ control signal. The memory typically takes 2 or 3 clock cycles to activate the selected row. (Fig. 4.5.3 shows 2 clock cycles). Then, the column address is latched by activating the $\overline{CAS}$ control signal. After a delay of one clock cycle, the memory places the first set of data bits on the data lines. The SDRAM then automatically increments the column address to access the next three sets of data bits in the selected row in three successive clock cycles.

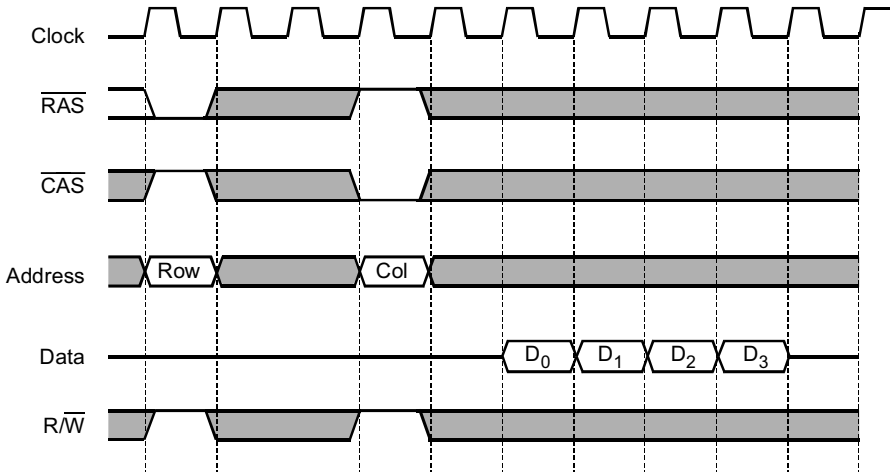


Fig. 4.5.3 Timing diagram of burst data transfer of length 4

The SDRAMs have built-in refresh circuitry. Its refresh counter provides the addresses of the rows that are selected for refreshing. In a typical SDRAM each row must be refreshed at least every 64 ms.

4.5.3.2 Performance Measures

The performance of the memory can be measured using two typical performance parameters : memory latency and bandwidth.

Memory latency : It is the term used to refer to the amount of time it takes to transfer a word of data to or from the memory. In case of single word transfer memory latency gives proper indication of its performance. In case of burst data transfer, number of data words are transferred and time needed to complete the operation depends also on the rate at which successive words can be transferred and on the rate at which successive words can be transferred and on the size of the block. In block transfers, the term memory latency is used to indicate the time it takes to transfer the first word of the block data. This time is usually substantially longer than the time needed to transfer each subsequent word of a block. Therefore, memory latency can not be a proper parameters to measure the perform of memory under block transfer of data.

Memory bandwidth : We know that the block size in block data transfer is not fixed, hence it is useful to define a performance measure in terms of the number of bits or bytes that can be transferred in one second. This performance measure is often referred to as the **memory bandwidth**. The bandwidth of a memory unit depends on the speed of memory access and on the number of bits that can be accessed in parallel. The number of bits that can be accessed in parallel depends on the data bus width of the processor. Therefore, the bandwidth is the product of the rate at which data are transferred (and accessed) and the width of the data bus.

4.5.3.3 Double-Data-Rate SDRAM

The outcome of continuous efforts taken to improve the memory performance is a faster version of SDRAM. In the timing diagram shown in Fig. 4.5.4 we have seen that the standard SDRAM performs its operations on the rising edge of the clock signal. On the other hand, the faster SDRAMs transfers data on both the edges of the clock signal. The latency of the faster SDRAMs is same as that for standard SDRAM. However, since they transfer data on both the edges of the clock signal, their bandwidth is effectively doubled for long burst transfer. Such faster SDRAMs are known as **Double-Data-Rate SDRAMs (DDR SDRAMs)**.

In DDR SDRAMs, the cell array is organized in two banks as shown in Fig. 4.5.4. As shown in Fig. 4.5.4 it has dual bank architecture to support on-chip parallelism. It also

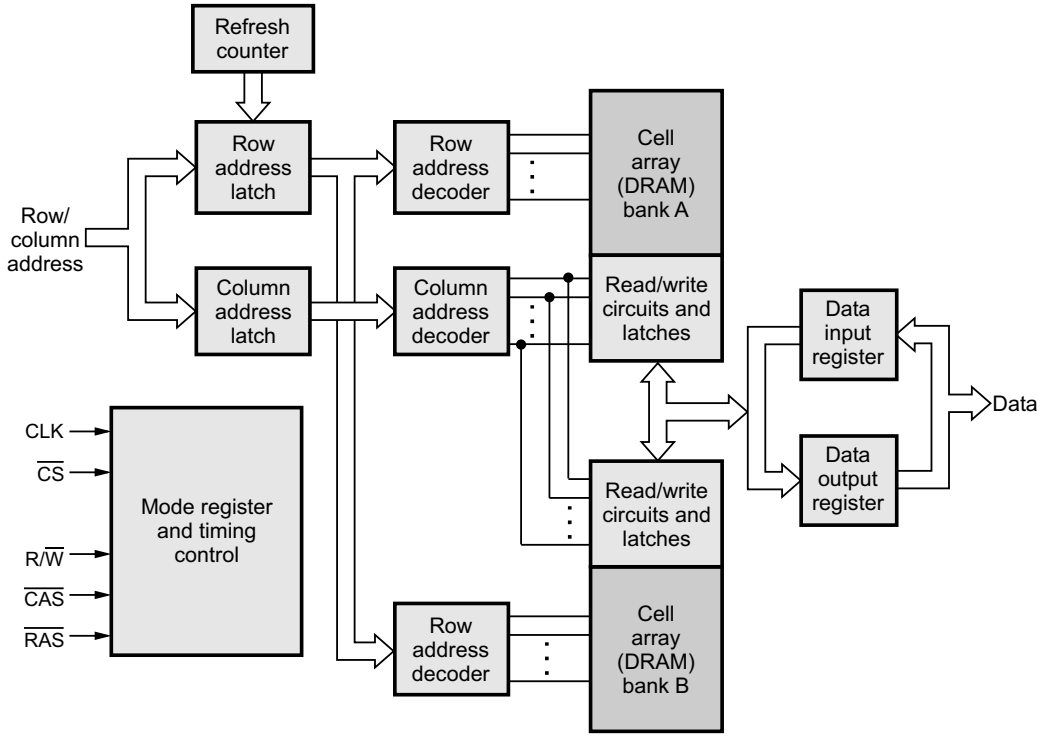


Fig. 4.5.4

supports brust mode transfer. This allows the sequential reading of number of bits from same row, eliminating the address setup time for each bit. Each bank can be accessed separately. Consecutive words of a given block are stored in different banks. Therefore, it is possible to transfer two words on sucessive edges of the clock signal.

4.5.4 Rambus DRAM

Rambus is small company founded in 1989 to develop a new high bandwidth DRAM interface. They designed vertical packages of Rambus DRAMs , with all pins on one side. Their design abandons the conventional pin-level interface : separate data bus and multiplexed address bus. A single byte-serial bus transfers control, addresses, as well as data. Rambus has following features :

- 500 Mbps data rates
- Vertically mounted DRAMs
- Physically constrained bidirectional bus
- Reduced voltage swing
- Synchronous design
- Dual bank row-caching

Rambus DRAMs provide high bandwidth with small number of chips, large row caches and good match for frame buffer design. But it has some disadvantages. These are :

- High latency
- Need to incorporate proprietary macros of bus interface

Byte serial to word parallel conversion delay.

4.5.5 Ramlink DRAM

Ramlink is a standard developed by the IEEE computer society. The working group applied the techniques adopted is Scalable Coherent Interface (SCI) to develop a ring-topology interconnect memory device. Ramlink also uses a high speed byte-wide bus, as in the case of rambus. However, instead of a single short bus, Ramlink using a ring topology with point-to-point links, as shown in the Fig. 4.5.5.

The advantage of Ramlink is that it is non-proprietary with a general interconnect. It does not have a physically constrained interconnect bus structure. The disadvantages of Ramlink are longer latency access and lack of availability.

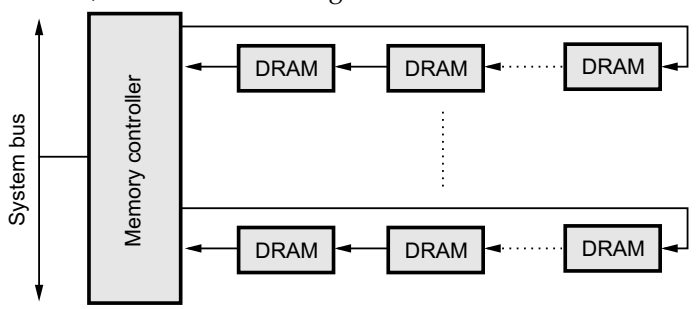


Fig. 4.5.5 RAM link architecture

4.5.6 EDORAM

EDORAM stands for Extended Data Out RAM. Extended Data Out offers high speed random access of memory cells within the same row. Fig. 4.5.6 shows typical organization for EDORAM.

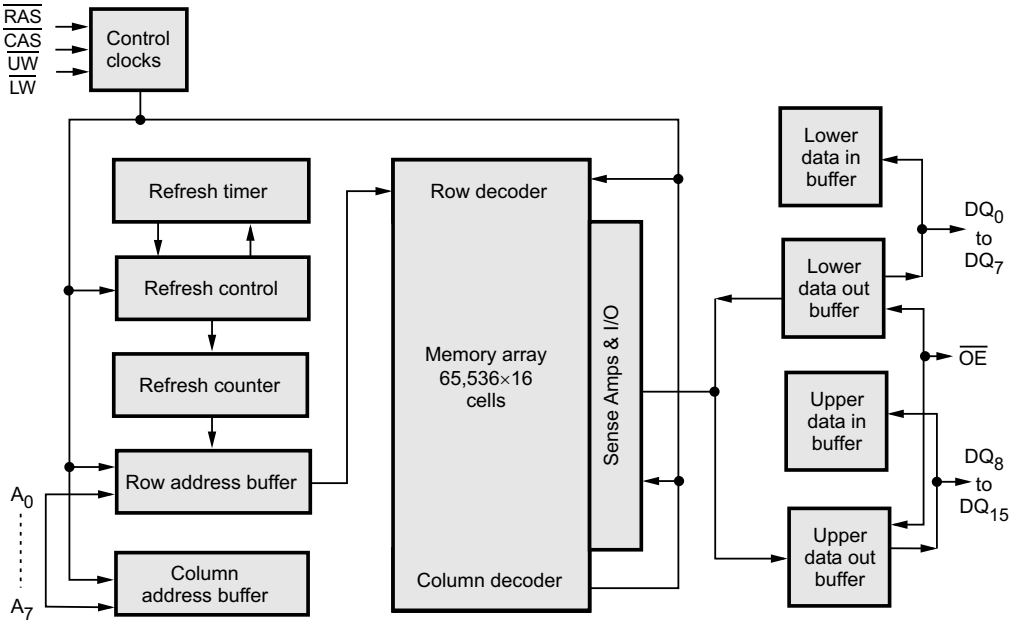


Fig. 4.5.6 Typical organization for EDORAM

Pin name	Pin function
$A_0 - A_7$	Address inputs
DQ_{0-15}	Data in/out
$\overline{RAS}$	Row address strobe
$\overline{CAS}$	Column address strobe
$\overline{UW}$	Upper write input
$\overline{LW}$	Lower write input
$\overline{OE}$	Data output enable

Review Questions

1. *Differentiate DRAM and SDRAM.* **Dec.-08, Marks 2**
2. *Compare SDRAM with DDR SDRAM.* **May-09, Marks 2**
3. *Write a note on synchronous DRAM.* **Dec.-10, Marks 8**

4.6 Serial Access Memories

May-07, 09, 10, Dec.-07, 08, 10, 11

In serial access memories data must be accessed in a predetermined order via read-write circuitry that is shared by different storage locations. Large serial access memories typically store information in a fixed set of tracks, each consisting of a sequence of 1-bit storage cells. A track has one or more access points at which a read-write head can read/write information to or from the track. A stored item is then accessed serially by moving either the stored information or the read-write heads or both.

The Fig. 4.6.1 shows the typical organization of a serial memory unit. It is assumed that each word is stored along a single track and that each access results in the transfer of a block of words. The address of the data to be accessed is sent to the address decoder, whose output determines the track to be used (the address of track) and the location of the desired block of information within the track (address of block). The address of track determines the particular read-write head to be selected. The read-write head is then moved to the position from where it can access the desired track.

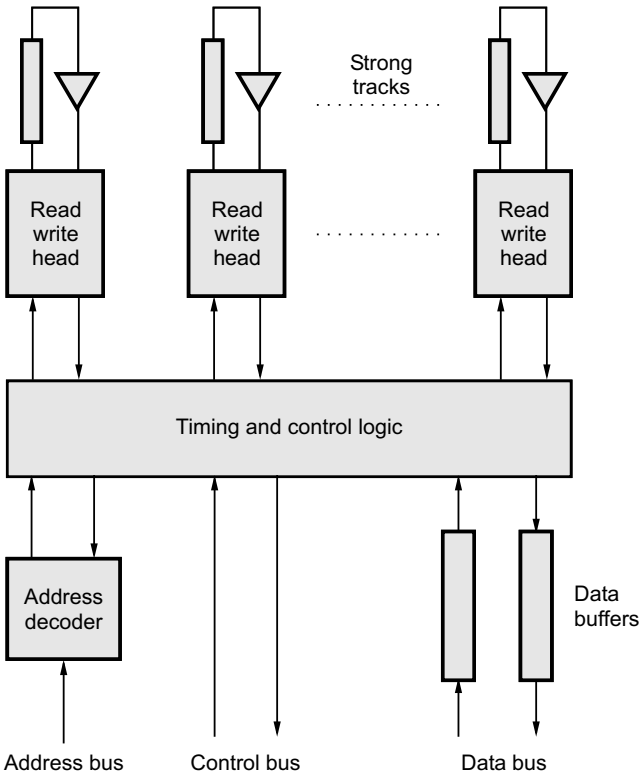


Fig. 4.6.1 Typical organization of serial-access memory unit

The read-write head is moved along the track to search for the desired block. A track position indicator generates the address of the block that is currently passing the read-write head. The generated address is compared with the block address produced by the address decoder. When they match, the selected read-write head is enabled and data transfer takes place.

Different types of storage media and access mechanisms are used to construct the serial access memories. Magnetic disk, magnetic-tape and optical memories fall into this category. Let us discuss these memories in detail.

4.6.1 Magnetic Disk Memory

4.6.1.1 Magnetic Surface Recording

A magnetic disk is a thin, circular metal plate. It is coated with a thin magnetic film, usually on both sides. Digital information is stored on the magnetic disk by magnetizing the magnetic surface in a particular direction as shown in the Fig. 4.6.2.

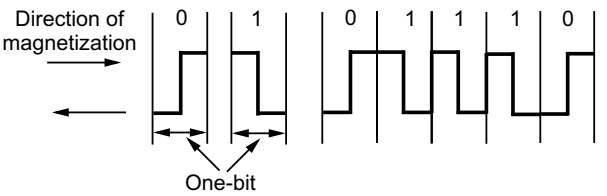


Fig. 4.6.2 Bit representation of magnetic disk

The disks are mounted on a rotary drive so that the magnetized surface moves in close proximity to magnetizing coil or head as shown in Fig. 4.6.3. The head consists of a magnetic yoke and the magnetizing coil, as shown in the Fig. 4.6.4. Digital

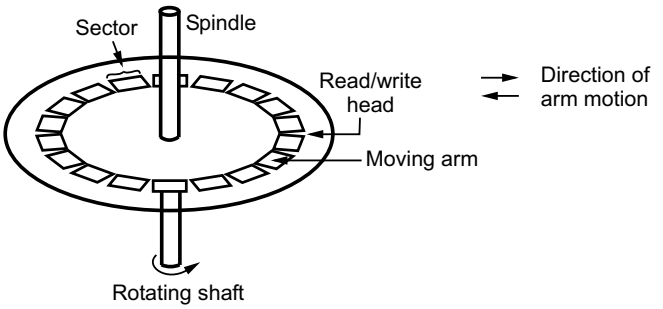


Fig. 4.6.3 (a) Rotating shaft

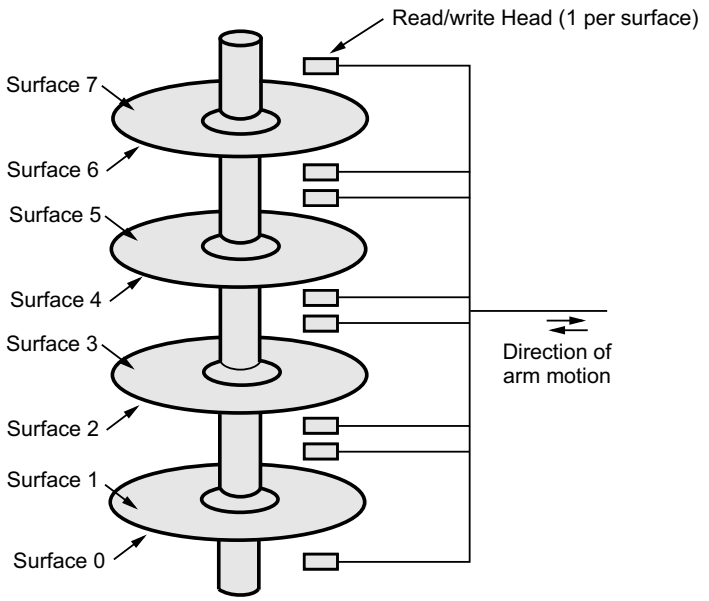


Fig. 4.6.3 (b) Mechanical structure

information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately below the head. The same head is used for reading the stored information. In this case, when the surface of the disk passes under the head, it generates a current and induces voltage in the coil of the same polarity as the one already recorded. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film.

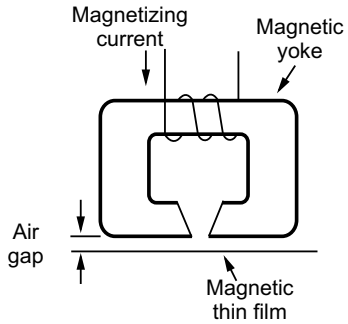


Fig. 4.6.4 Read/Write head detail

4.6.1.2 Data Organization and Formatting

The data on the disks is organized in a concentric set of rings as shown in the Fig. 4.6.5. These concentric set of rings are called as **tracks**. Each track has a same width as head and adjacent tracks are separated by **gaps**. The gap between two adjacent tracks prevents, or at least minimizes, errors due to misalignment of the head or simply interference of magnetic fields. Data bits are stored serially on each track. The same numbers of bits are stored on each track. Thus, the density, in bits per linear inch,

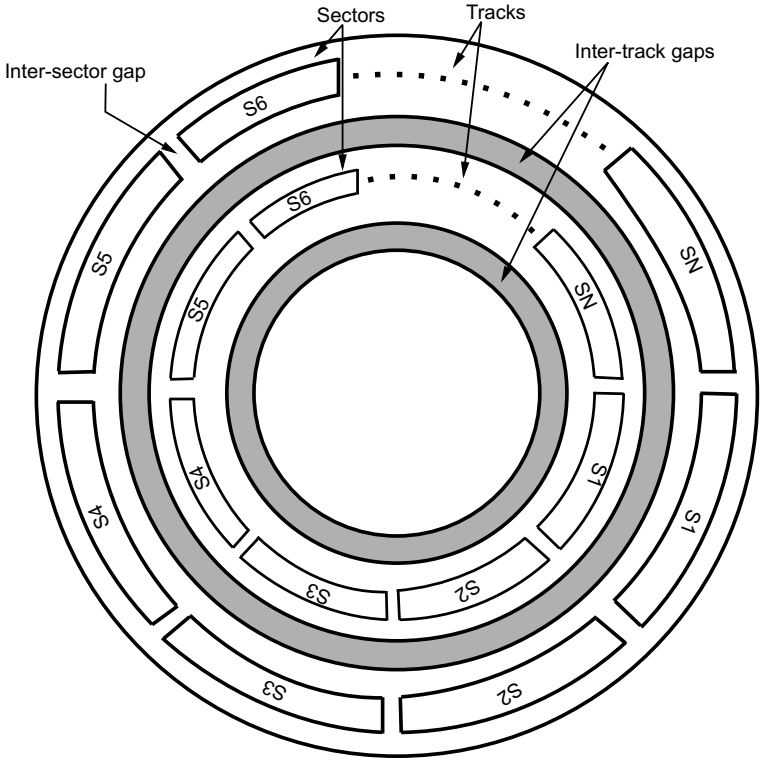


Fig. 4.6.5 Disk data layout

increases as we move from the outermost track to the innermost track. The tracks are further divided into sectors, as shown in the Fig. 4.6.5. Each sector stores a block of data which can be transferred to or from the disk. To avoid magnetic interference between two adjacent sectors the gap is introduced between two adjacent sectors.

Data on disk is addressed by specifying the disk number or head number, track number and the sector number. The start and end of each sector is determined by the control data stored on each sector.

Fig. 4.6.6 shows disk format for disk. As shown in the Fig. 4.6.6, each track contains 30 fixed length sectors of 600 bytes each. Each sector holds 512 bytes of data plus control information useful to the disk controller. The control information is represented by ID field, which is used as an unique identifier to locate a particular sector.

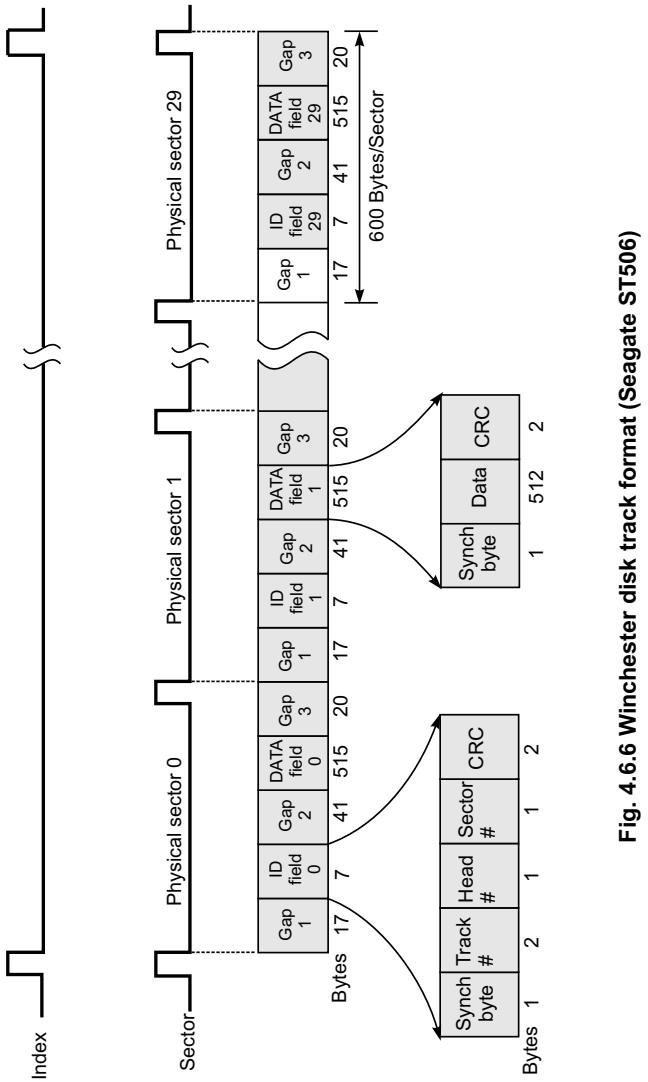


Fig. 4.6.6 Winchester disk track format (Seagate ST506)

4.6.1.3 Characteristics

The Table 4.6.1 lists the important characteristics of a magnetic disk.

Attributes	Type
Head motion	Fixed head (one per track), Movable head (one per surface)
Disk portability	Nonremovable disk, Removable disk
Sides	Single - sided, Double - sided
Disk/surface	Single - surface, Multiple - surfaces
Head mechanism	Contact fixed gap, Aerodynamic gap

Table 4.6.1

In fixed head disk, there is one head per track. These heads are mounted on a rigid arm as shown in the Fig. 4.6.3. In a movable head disk, there is only one head and it is moved between tracks by control mechanism. The nonremovable disk is permanently mounted in the disk drive whereas removable disk can be removed and replaced with another disk.

In single sided disk, the magnetic coating is applied on only one side of the disk. On the other hand, in double sided disk it is on both the sides of disk, doubling the storage capacity.

Some disk drives accommodate multiple surfaces stacked vertically apart with separate pair of heads as shown in the Fig. 4.6.3.

Finally, disk drives are also classified according to head positions : Contact, fixed gap or aerodynamic gap.

4.6.1.4 Typical Disks

The specifications of some common commercially available disk drives are shown in Table 4.6.2.

Capacity	Cylinders	Heads	Sectors	Bytes/Sector
4.302 GB	523	255	63	512
2.159 GB	523	128	63	512
1.08 GB	523	64	63	512

Table 4.6.2

4.6.1.5 Access Time

The disk access time is the time delay between receiving an address and the beginning of the actual data transfer. The disk access time has two time components :

seek time and rotational delay. The **seek time** is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. The average seek time is in the range of 5-8 ms. The **rotational delay** also known as **latency time** is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed section passes under the read/write head.

4.6.1.6 Data Buffer/Cache

A disk drive is usually connected to the computer using standard interconnection scheme called **SCSI bus**. A disk drive along with the required SCSI interface circuitry is known as **SCSI drive**. The transfer speed of SCSI bus is much higher than the rate at which data can be read from the disk tracks. To adjust the possible differences in transfer rates between the disk and the SCSI bus, data buffer is included in the disk unit. This data buffer is a semiconductor memory, capable of storing a few megabytes of data. The requested data from the disk is transferred first to the buffer at the rate dependent on the access time of the disk. Then the data from the buffer is transferred to other devices connected to the SCSI bus, normally the main memory at the maximum rate allowed by the SCSI bus.

The data buffers also serves as a **disk cache**. When a read request arrives at the disk, the disk controller checks to see if the desired data are already available in the data buffer. If so, the data can be accessed and placed on the SCSI bus saving the disk access time.

4.6.1.7 Disk Controller

Disk controller is the hardware interface provided to control the operation of a disk drive. The disk controller can be used to control more than one drive. The Fig. 4.6.7 shows a connections of disk controller which controls two disk drives.

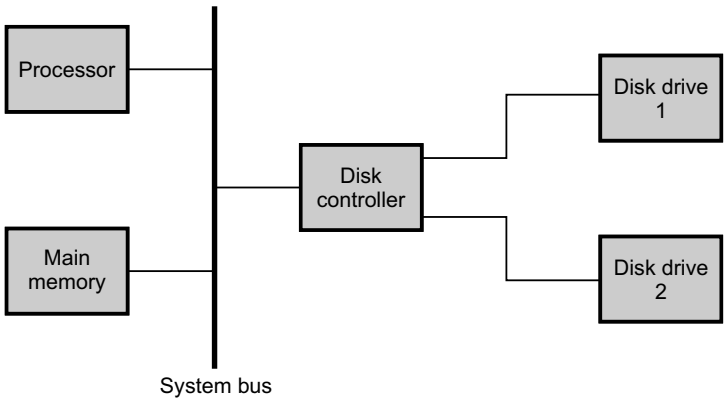


Fig. 4.6.7 Disk controller interface connections

The disk controller used in PCs also can drive two disk drives. It is either connected directly to the processor system bus, or to an expansion bus such as PCI. The disk controller transfers data from and to the main memory on a DMA basis. Actually, these transfers are to or from the data buffer, which is implemented as a part of the disk controller module.

Write operation : In write operation, the data bytes received from the main memory in DMA mode are stored in the data buffer. The disk controller takes these bytes from the data buffer, serializes them into data bits, mixes them with clock bits and sends the MFM write data to the disk drive.

Read operation : In read operation, the disk controller read MFM data from disk drive. The MFM data consists of DATA bits and clock bits. The disk controller separates them, converts the data bits into parallel bytes and stores them in the data buffer. From the data buffer, these data bytes are transferred to main memory in DMA mode.

The Error Correcting Code (ECC) logic generates 32-bit ECC pattern for the data bytes, which is also written in the sector along with the data bytes. During read operation, the disk controller checks the ECC pattern to verify whether the data bytes are error-free. If there is error, disk controller corrects it. The ECC is used for the data field in each sector whereas CRCC (Cycles Redundancy Characters Code) is used for the ID field in each sector. The retry logic is used to retry an operation on encountering an error. The diagnostic logic performs various tests on disk controller circuit to detect malfunctions. It also test the disk media.

We can summarise the major function of disk controller as follows :

- **Seek :** It causes the disk drive to move the read/write head from its current position to the specified track.
- **Read :** It initiates a read operation, starting at the address specified in the disk address register. It converts serial data bits from the disk into parallel bytes and stores them in the data buffer for transfer to main memory. The number of bytes/words to be transferred is determined by the word count register.
- **Write :** It receives data bytes from the main memory and stores them into data buffer. It then serializes these data bytes into data bits, mixes them with the clock bits and sends to the drive.
- **Error checking :** It computers ECC value for the data read from the disk and compares it with the corresponding ECC value read from the disk. In case of error, it corrects the error if possible; otherwise, it activates the retry logic to repeat the read the process.

4.6.1.8 Loading of Operating System from Disk

Disk data consists of various application softwares along with the operating system. When power is turn on the operating system is loaded from the disk into the main memory. This process takes place as a part of **booting**. The booting process is initiated by the monitor program stored in the nonvolatile ROM. This monitor program first loads the block called **boot block** or **boot record** from the disk at address 0. This block contains the loader program, which is used to load part of operating system from disk into the main memory.

Example 4.6.1 *A Winchester magnetic disk unit has densities of 40×10^6 bits per square inch of surface.*

i) If the inner diameter of recording area is 4 inches and the outer diameter of 7 inches, what is the average bit density along a track if radial track spacing density is 2000 tracks/inch.

ii) What is the data transfer rate in bytes/sec at a rotational speed of 3600 RPM ?

Solution :

$$\text{Inner diameter } D_I = 4 \text{ inches}$$

$$\text{Outer diameter } D_O = 7 \text{ inches}$$

$$\text{Density} = 40 \times 10^6 \text{ bit/inch}^2$$

i) $$\text{Total recording area} = \pi \times \left(\frac{D_O}{2}\right)^2 - \pi\left(\frac{D_I}{2}\right)^2 = 25.91814 \text{ inch}^2$$

$$\text{Total disk capacity} = 25.91814 \times 40 \times 10^6$$

$$= 103.67 \times 10^7$$

$$\text{Radia track spacing} = 7 - 4 = 3$$

$$\text{Number of tracks on the disk} = 2000 \times 3 = 6000 \text{ / inch}$$

$$\text{Average bit density along a track} = \frac{103.67 \times 10^7}{6000}$$

$$= 172.78 \text{ kbits/track}$$

ii) $$\text{Data transfer rate in bytes / sec} = \text{Average bit density along a track} \times \text{Rotation speed}$$

$$= \frac{172.78\,K}{8} \times \frac{3600}{60}$$

$$= 1.2959 \text{ Mbytes/sec}$$

4.6.2 Floppy Disk Memory

The magnetic disks that we described above are called hard-disks. These disks, in the form of disk - packs are suitable for storage of large amount of data. **Floppy** disks are smaller, simpler and cheaper disk units that consist of a flexible, removable, plastic diskettes coated with magnetic material. The diskette is enclosed in a plastic jacket, which has an opening where the head makes contact with the diskette. A hole in the diskette allows a spindle mechanism in the disk drive to position and rotate the diskette as shown in the Fig. 4.6.8.

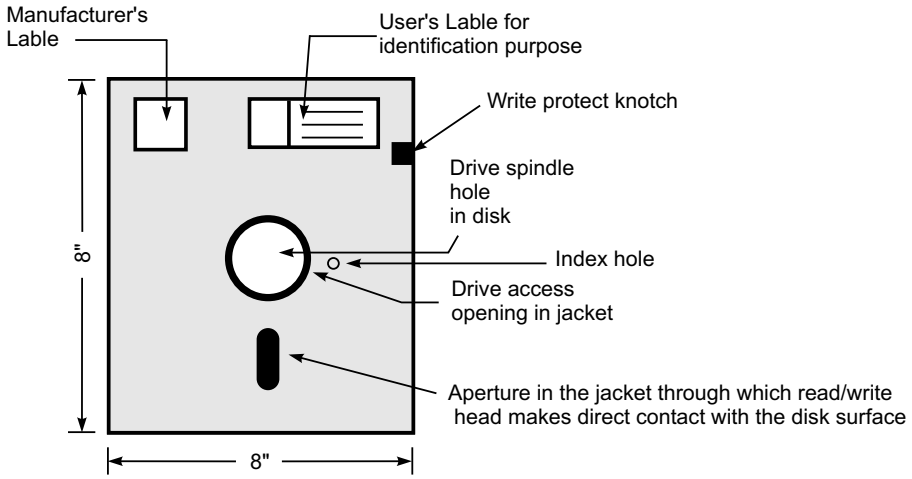

Manufacturer's Lable
User's Lable for identification purpose
Write protect knotch
Drive spindle hole in disk
Index hole
Drive access opening in jacket
Aperture in the jacket through which read/write head makes direct contact with the disk surface
8"
8"

Fig. 4.6.8

4.6.2.1 Specifications of Floppy Disks

Physical specifications	Capacity
5.25 inch floppy disk, 1 sided	180 kB
5.25 inch floppy disk, 2 sided	360 kB
5.25 inch floppy disk, 2 sided	1.2 MB
3.5 inch floppy disk, 2 sided	1.44 MB

Table 4.6.3

4.6.2.2 Disk Format

The data on the disks is organised in a concentric set of rings. These concentric set of rings are called as tracks. Each track has same width as head and adjacent tracks are separated by gaps. The gap between two adjacent tracks prevents, or at least minimizes errors due to misalignment of the head or simply interference of magnetic fields. Data bits are stored serially on each track. The same number of bits are stored in each track. The same number of bits are stored on each track. Thus, the density in bits per linear inch increases as we move from the outermost track to the innermost track. The tracks are further divided into sectors. In a PC, the DOS organizes 9 sectors (8 sectors in old

DOS version) on each track. Each sector stores 512 bytes of data which can be transferred to or from the disk. To avoid magnetic interference between two adjacent sectors, a gap is introduced between two adjacent sectors.

Fig. 4.6.9 shows IBM 3740 floppy disk track format. As shown in the Fig. 4.6.9, each sector consists of ID field and a data field. The ID field contains the address of the sector. The data field contains the actual data stored in the sector. As we can see, besides the bytes used to store data, many bytes are used for track and sector identification, synchronization, error checking and buffering between sectors. Address marks shown at several places in this format are used to identify the start of the field. (Fig. 4.6.9 see on next page)

4.6.2.3 Storage Density

The IBM 3740 format shown in Fig. 4.6.9 is set up for single density recording. An 8-in disk in this format has one index track and 76 data tracks. Since each track has 26 sectors with 128 data bytes in each sector, the total is about 250 kbytes. If double density recording is used, the capacity increases to about 500 kbytes. Table 4.6.4 shows the number of tracks, number of sectors and some other information for floppy disks commonly used with IBM PC.

Parameter	$3\frac{1}{2}$ Inch disk		$5\frac{1}{4}$ Inch disk	
	1.44 MB HD	720 K LD	1.2 MB HD	360 K DSDD
Sectors per track	18	9	15	9
Total number of sectors	2880	1440	2400	720
Number of tracks	80	80	80	40
Sectors per cluster	1	2	1	2
Allowable entries in root directory	224	112	224	112

Table 4.6.4 Floppy disks tracks, sectors and capacities

In IBM 3740 floppy disk format, in each sector 2 bytes are used to store CRC error checking code. These are used to check for errors when the ID and the data are read out. One way the 2 CRC bytes can be produced is to treat the 128 data bytes as a single large binary number and divide this number by a constant. The 16-bit remainder from this division is written in after the data bytes as the CRC bytes. When the data bytes and the CRC bytes are read out, the CRC bytes are subtracted from the data string. The result is divided by the original constant. Since the original remainder has already been subtracted, the result of the division should be zero if the data was read out correctly.

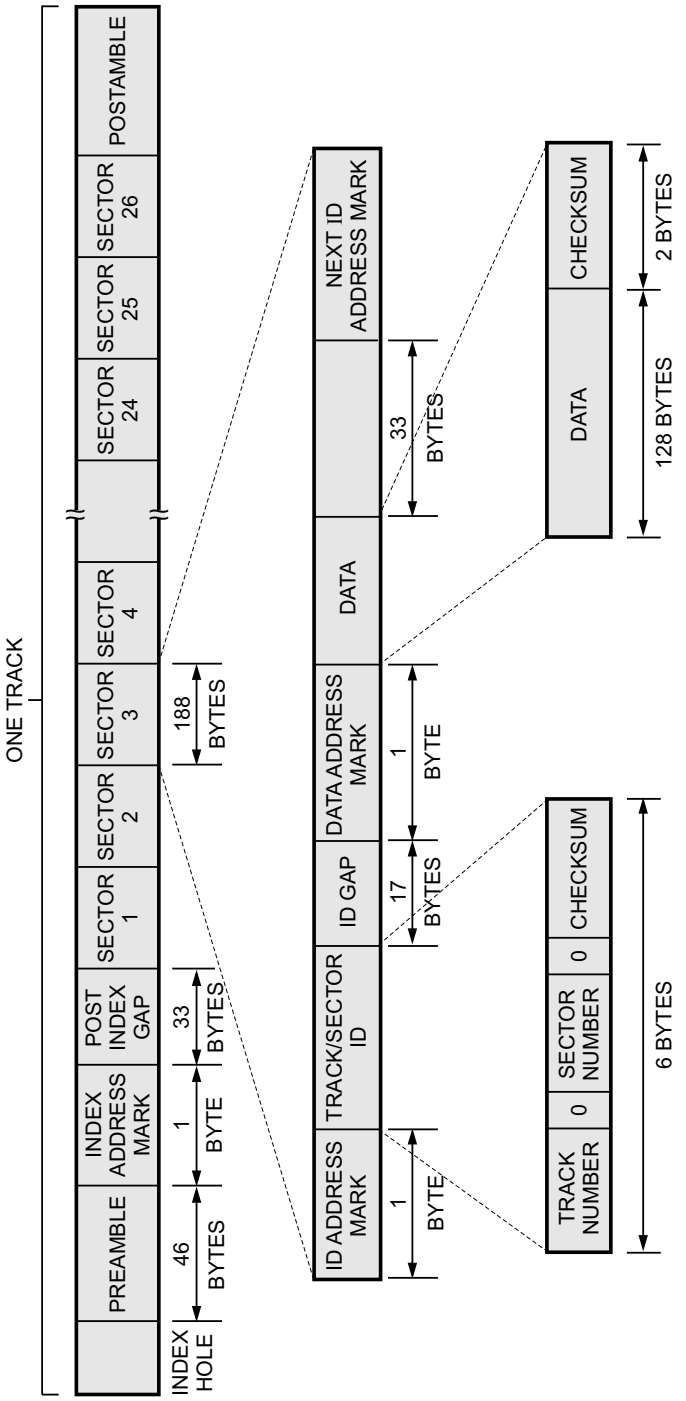


Fig. 4.6.9 IBM 3740 floppy disk sectored track format

4.6.3 Magnetic Tape

Magnetic tape is one of the most popular storage medium for large data that are sequentially accessed and processed. The tape is formed by depositing magnetic film on a very thin and 1/2 inch or 1/4 inch wide plastic tape. Usually, iron oxide is used as a magnetizing material. The tape ribbon itself is stored in reels similar to the tape used on a tape recorder except that it is of high quality and more durable. Like audio tape, computer tape can be erased and reused indefinitely. Old data on a tape are automatically erased as new data are recorded in the same area.

The information is recorded on the tape with the help of read/write head. It magnetizes or nonmagnetizes tiny invisible spots (representing 1's and 0's) on the iron oxide side of the tape, as shown in the Fig. 4.6.10.

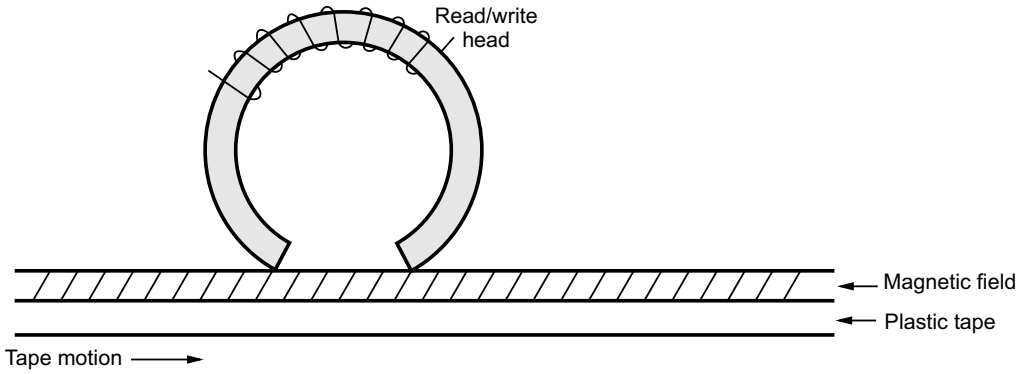


Fig. 4.6.10 Magnetic recording with read/write head

Usually, seven or nine bits (corresponding to one character) are recorded in parallel across the width of the tape, perpendicular to the direction of motion. A separate read/write head is provided for each bit position on the tape, so that all bits of characters can be read or written in parallel. One of the character bit is used as a parity bit. This is illustrated in Fig. 4.6.11.

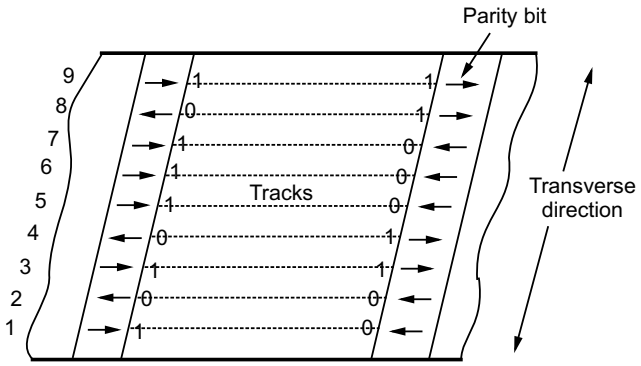


Fig. 4.6.11 Magnetic recorded tape

Data on the tape are organized in the form of *records* separated by gaps, as shown in the Fig. 4.6.12. A set of related records constitutes a *file*. A *file mark* is used to identify the beginning of the file. The file mark is a special single or multiple character record,

usually preceded by a gap longer than the interrecord gap. The record following the file mark (first record) can be used as a *header* or *identifier* for this file. This allows user to identify a particular file from number of files recorded on the tape.

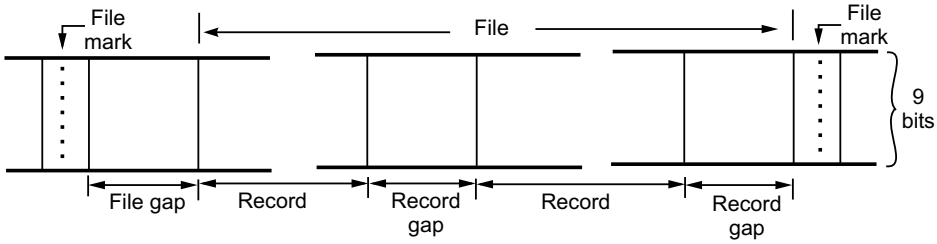


Fig. 4.6.12 Organization of data on magnetic tape

We know that, the data on a nine track tap is stored in parallel consists of data byte and a parity bit. Now-a-days tapes are available with several hundred tracks. In which, a read-write head can simultaneously access all tracks. Data transfer takes place when the tape is moving at constant velocity relative to a read-write head; hence the maximum transfer data-transfer rate depends largely on the storage density along the tape and the speed of the tape.

Example 4.6.2 *For 120 track tape with a storage density per track of 100 kB/in and tape speed of 50 in/s, calculate the maximum data transfer rate. If tape length is 450 feets, calculate the storage capacity of the tape.*

Solution : The maximum data transfer rate is given by

$$d = \text{Storage density/Track} \times \text{Tape speed} \times \frac{\text{Number of tracks}}{8}$$

$$= 100 \times 10^3 \times 50 \times 120/8 = 75 \text{ Mbytes/sec}$$

The storage capacity of the tape is given by

$$S = \frac{d \times \text{Tape length in inches}}{\text{Tape speed}}$$

$$= \frac{75 \text{ Mbytes / sec} \times (450 \times 12)}{50 \text{ in / sec}} = 8.1 \text{ Gbyte}$$

The information on the magnetic tape is organised into blocks. These blocks have a fixed length and separated by inserting gap between them. This gap allows the tape to start and stop between blocks. If the block length is B_L and the inter block gap length is G_L, then the utilization factor of tape, u is given by

$$u = \frac{B_L}{B_L + G_L}$$

Example 4.6.3 *Calculate the utilization factor of tape, if gap length is 0.5 in, the storage density S = 3000 bytes/in and block storage capacity is 6 kbytes.*

Solution : From the given data the block length can be given as

$$B_L = \frac{\text{Block storage capacity}}{\text{Storage density}} = \frac{6 \times 10^3}{3000} = 2 \text{ inches}$$

Therefore, the utilization factor of tape,

$$u = \frac{2}{2 + 0.5} = 0.8$$

Example 4.6.4 *A magnetic tape system accommodates 2400 ft reels of standard ninetrack tape. The tape is moved past the recording head at rate of 200 in/sec.*

i) What must the linear tape-recording density be in order to achieve data transfer rate of 10 Mbits/sec ?

ii) Suppose that the data on the tape is organized into blocks each containing 32 kbytes. A gap of 0.3 in separates each block. How many bytes may be stored on the tape ?

Solution : Given : Tape speed = 200 in/sec

Data transfer rate = 10 Mbits/sec

i) From the given data, the tape recording density to achieve data transfer rate of 10 Mbytes/sec is given as

$$\text{Tape recording density} = \frac{10 \text{ Mbits / sec}}{200 \text{ in / sec}} \times \frac{1}{8}$$

$$= 6.25 \text{ kbytes/in}$$

ii) The block length can be given as

$$B_L = \frac{\text{Block storage capacity}}{\text{Storage density}} = \frac{32 \text{ kbytes}}{6.25 \text{ kbytes / in}}$$

$$= 5.12 \text{ in}$$

$$\text{The number of block on the tape} = \frac{\text{Tape length}}{B_L + G_L} = \frac{2400 \times 12}{5.12 + 0.3}$$

$$= 5313.6 \approx 5313$$

Therefore, storage capacity of tape is

$$S = \text{Number of blocks} \times \text{Storage capacity of block}$$

$$= 5313 \times 32 \text{ kbytes} = 170.016 \text{ Mbytes}$$

Due to the inter block gap and the time needed to start and stop the tape between accesses, the effective data transfer rate d_{eff} is actually less than the maximum rate d. If we assume, t_D be the time to scan a data block, t_G be the time to scan the interblock gap, and t_{SS} be the time to start and stop the tape, then

$$d_{eff} = \frac{t_D \times d}{t_D + t_G + t_{SS}}$$

The *tape drive* is used for tape movement and it is controlled by the *tape controller*. The tape controller issues special commands for the operation of tape. These are as follows :

1. Read
2. Write
3. Rewind
4. Rewind and unload
5. Erase tape
6. Write tape mark
7. Forward space one record
8. Backspace one record
9. Forward space one file
10. Backspace one file

The tape mark is used to identify the beginning of the file whereas the end of the tape is identified by the EOT (End of Tape) character.

There are two formatting methods used with the magnetic tapes. In the first method, the records are variable in length. This allows efficient use of the tape, but it does not permit updating or overwriting of records in place. In the second method, fixed length records are used. In this case, it is possible to update records in place.

The magnetic tapes are commonly used for backing up information on magnetic disk.

Advantages of Magnetic Tape

1. **Unlimited storage :** The storage capacity of magnetic tape is virtually unlimited because we can use as many tapes as required for recording.
2. **High data density :** A typical 10.5 inch reel of magnetic tape is 28800 inches long and is able to hold upto 6250 characters per inch. Therefore, the maximum capacity of magnetic tape is about 180 million characters.
3. **Low cost :** The cost of magnetic tape is much less as compared to other storage devices.
4. **Rapid transfer rate :** Data transfer rate for a magnetic tape can be in excess of 1 million bytes per second.

5. **Ease of handling :** Since it is compact and light weighted it is easier to use it.
6. **Portability :** The tape is wound on a reel and it is packed in a plastic package, hence it is very convenient way of carrying information from one place to another.

Disadvantages of Magnetic Tape

1. **Sequential access :** Magnetic tape is a sequential access device and hence data recorded on tape cannot be accessed directly. Thus more time is required for accessing the record.
2. **Environmental restriction :** The magnetic tapes suffer from dust and uncontrolled humidity or temperature levels. This may cause data reading errors.

Comparison between magnetic disk and magnetic tape :

Attribute	Magnetic tape	Magnetic disk
Access	Sequential	Direct or sequential
Access time	More	Less
Environmental restrictions	More	Less
Data transfer rate	Comparatively less	More
Reliability	Less	More
Portability	More	Less
Cost	Less	High
Application	Used for backups	Used as a secondary storage device in computer systems

4.6.4 RAID

The magnetic disk systems just described provide a single, serial bit stream during data transfer. Recent disk - system developments have been directed at decreasing access time and increasing bandwidth during data flow through the use of multiple disks operating parallel. In multiple disk system, arrays of inexpensive disks operate independently and in parallel to improve the speed of data transfer. This system is referred to as RAID (Redundant Arrays of Inexpensive Disks). In a RAID system, a single large file is stored in several separate disk units by breaking the file up into a number of smaller pieces and storing these pieces on different disks. This is called **data stripping.** When the file is accessed for a read, all disks deliver their data in parallel. The total file transfer time can be given as, the transfer time that would be required in a single disk system divided by the number of disks used in the array. However, the time required to locate the beginning of the data on each disk, **access time**, is not reduced.

We have seen in RAID systems, a single file modules can be accessed simultaneously. This approach improves the speed of data transfer. The another approach in RAID

system is to add redundancy to improve reliability. This can be achieved by duplicating all pieces of the distributed, or striped, file on addition disk units, thus doubling the total number of disks used. With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. These are called as **RAID levels.** RAID levels do not imply a hierarchical relationship but designate different design architectures.

RAID Level 0

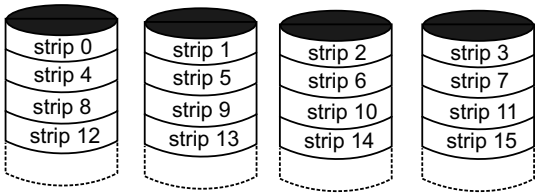


Fig. 4.6.13 RAID level 0 (Non-redundant)

Fig. 4.6.13 shows the data stripping used in the RAID level 0. The strips are mapped round-robin to consecutive array members. A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a **stripe.** In an n-disk array, the first n logical strips are physically stored as the first strip on each of the n disks, the second n strips are distributed as the second strips on each disk and so on. With this design architecture, two different I/O requests for two different blocks of data can be processed in parallel, reducing the I/O queuing time. The RAID level 0 architecture achieves the parallelism but it does not include redundancy to improve reliability. Hence, we can say that RAID level 0 is not a true member of the RAID family.

RAID Level 1

Fig. 4.6.14 shows how the data stripping is used and redundancy is achieved in RAID level 1. In RAID level 1, the redundancy is achieved by just duplicating all the data. The data stripping is used, same as in RAID level 0. But in this case, each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data.

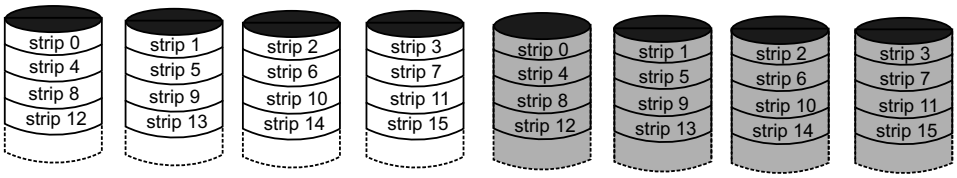


Fig. 4.6.14 RAID level 1 (Mirrored)

RAID Level 2

Fig. 4.6.15 shows the design architecture for RAID level 2. Here, data stripping used, same way as in RAID level 0 and RAID level 1. But to achieve reliability, an error correcting code is calculated across corresponding bits on each data disks, and the bits

of the code are stored in the corresponding bit positions on multiple parity disks. Usually hamming code is used as single error corrector and double error detector (SEC - DED).

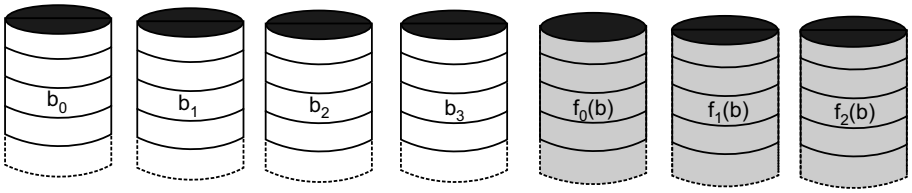


Fig. 4.6.15 RAID level 2 (Redundancy through hamming code)

RAID Level 3

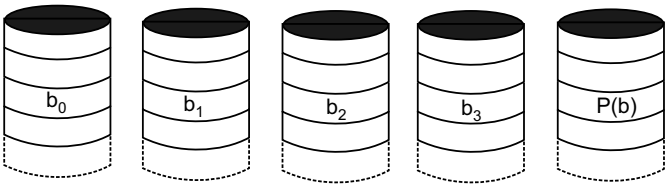


Fig. 4.6.16 RAID level 3 (Bit-interleaved parity)

Fig. 4.6.16 shows the design architecture for RAID level 3. Here, data bits are organized in similar fashion to RAID level 2. The difference is that RAID level 3 requires only a single redundant disk, no matter how large the disk array. The data stripping is used, similar to the other RAID levels. In RAID level 3 instead of an error correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

RAID Level 4

Fig. 4.6.17 shows the design architecture for RAID level 4. Here, data stripping used is same as used in other levels. But the size of strip is large which represents one data block. In RAID level 4, a bit by bit parity strip is calculated across corresponding data blocks on each data disk and the parity bits are stored in the corresponding strip on the parity disk. For each read/write operation parity bits are checked for data reliability. As a result, parity disk becomes a bottleneck.

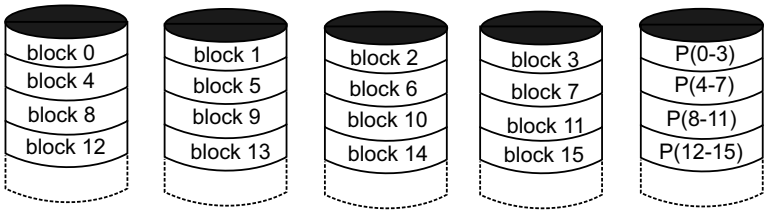


Fig. 4.6.17 RAID level 4 (Block level parity)

RAID Level 5

Fig. 4.6.18 shows the design architecture for level 5. Here, the data bits are organized in a similar fashion to RAID level 4. The difference is that RAID 5 distributes the parity strips across all disks. Due to the distribution of parity strips, the bottleneck found in RAID level 4 can be avoided.

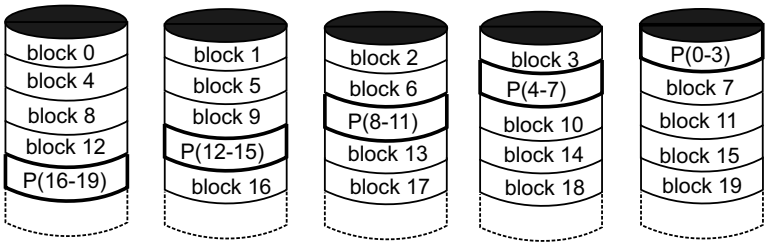


Fig. 4.6.18 RAID level 5 (Block-level distributed parity)

In RAID levels from 1 to 5, the redundant disk capacity is used to either duplicate all data or to store parity information which guarantees data recoverability in case of a disk failure.

4.6.5 Optical Memories

In 1983, the first optical memory device, Compact Disk (CD) was successfully launched. The CD is a nonerasable disk that can store more than 60 minutes of audio information on one side. The huge success of CD encouraged the development of optical storage technology and developers came out with three basic types of optical disk :

- Compact Disk Read-Only Memory (CD-ROM)
- Write-Once Read-Many (WORM) and
- Erasable Optical Disk

4.6.5.1 CD-ROM

The Compact Disk (CD) and Compact Disk Read Only Memory (CD-ROM) uses the similar technology. However, the CD-ROM players are more rugged and have error correction devices to ensure that data are properly transferred from disk to computer. Both disks consists of a rotating disk which is coated with a thin highly reflective material. In this system, the data recording is done by focusing a laser beam on the surface of the spinning disk, as shown in the Fig. 4.6.19.

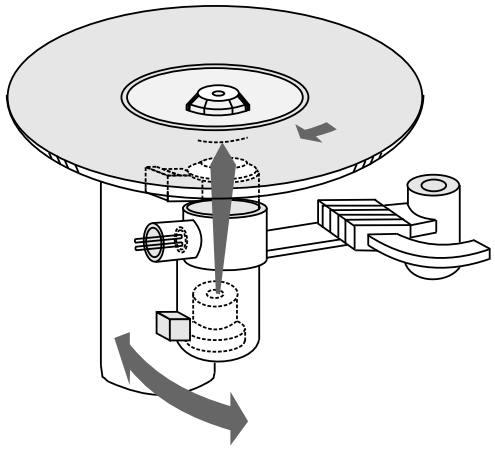


Fig. 4.6.19 Compact disk with laser beam control mechanism

The laser beam is turned ON and OFF according to digital signal. Because of which tiny holes (pits) are brunt into the metal coating of the disk along its tracks, as shown in the Fig. 4.6.20.

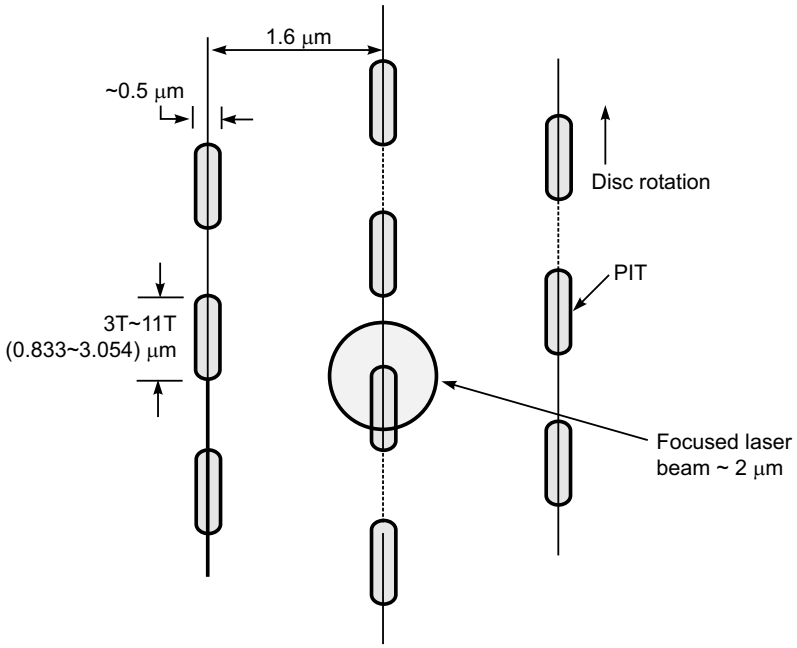


Fig. 4.6.20 Recorded portion of CD showing pits along its tracks

The data is recovered from the compact disk with an optical pickup, which moves across the surface of the rotating disk. Fig. 4.6.21 shows optical pickup system.

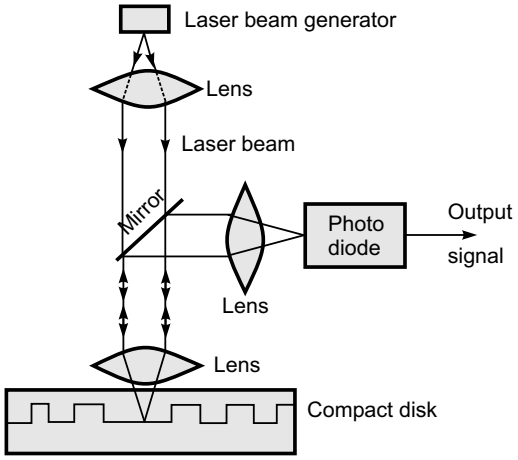


Fig. 4.6.21 Optical pickup system

Optical pickup system uses a semiconductor laser with approximately 5 milliwatt optical output irradiating a coherent beam with a 790 nanometer wavelength. This beam is incident on the compact disk through a half silver mirror. The mirror allows the beam to pass through itself but does not allow the returning beam to pass. When beam strikes a band interval between two pits, the light is almost totally reflected. When it strikes a pit a lower intensity light is returned. The change in intensity represents 1 and unchanged intensity represents 0.

A pit near the center of a rotating disk travels slower than a pit on the outside. It is necessary to compensate for the variation in speed so that the laser can read all the pits at the same rate. This can be done by increasing the spacing between bits of information recorded in segments of the disk. Thus the information can be read at the same rate by rotating the disk at a fixed speed, known as the **Constant Angular Velocity (CAV).** The Fig. 4.6.22 shows the layout of a disk using CAV. As shown in the Fig. 4.6.20, the disk is divided into a number of pie-shaped sectors and into a series of concentric tracks. This arrangement makes access to CD-ROM very easy. It is possible to access CD-ROM by specifying particular track and sector. However, due to this arrangement a lot of space in the outer track is wasted.

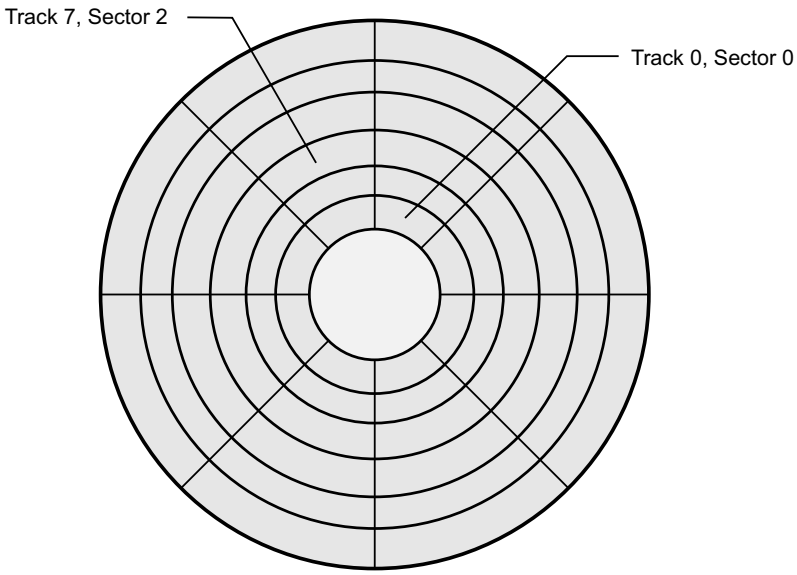


Fig. 4.6.22 Disk layout using constant angular velocity

Due to this disadvantage of CAV method, now-a-days **Constant Linear Velocity (CLV)** method is used to put information on the disk. In this method, the information is packed evenly across the disk in segments of the same size and these are scanned at the same rate by rotating the disk at a variable speed. The Fig. 4.6.23 shows the layout of a disk using CLV.

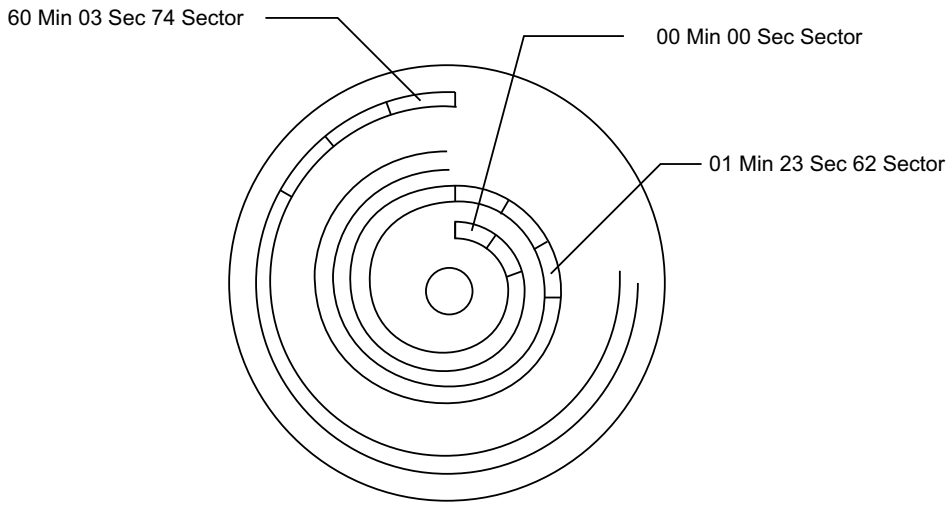


Fig. 4.6.23 Disk layout using constant linear velocity

Fig. 4.6.24 shows the physical characteristics of a compact disk. It has an outer diameter of 120 millimeters, a center hole 15 millimeters across, a thickness of 1.2 millimeters and a weight of about 14 grams. The digital audio signal is recorded in the form of pits on the CD. Each data pit is about 0.5 μm wide. Its length and the length of the land separating each pit varies from 0.833 to 3.054 μm. It is that varying ratio of pit and land that encodes the data itself. Pitch of the tracks (separation between adjacent tracks) is 1.6 μm, as shown in the Fig. 4.6.24. Such compact disk can store upto 650 Mbytes of data.

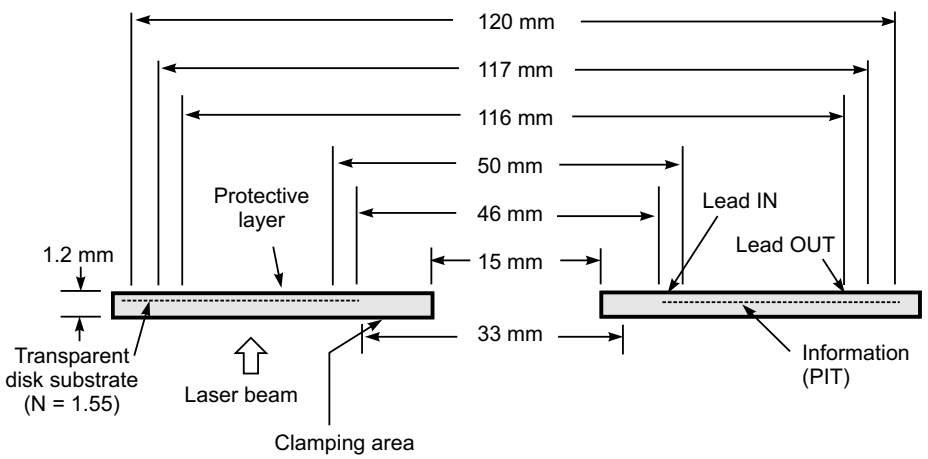


Fig. 4.6.24 Physical characteristics of compact disk

The Fig. 4.6.25 shows the format for typical data block on the CD-ROM. Such blocks are sequentially stored on the CD-ROM. Each data block consists of following fields :

SYNC : This field identifies the beginning of a block. It consists of one byte of all zeroes, 10 bytes of all ones and then one byte of all zeroes.

HEADER : This field consists of address and mode information. Mode 1 indicates the blank data field, Mode 2 specifies the use of an error-correcting code and 2048 bytes of data and Mode 2 specifies 2336 bytes of user data with no error correcting code.

DATA : This field stores user data.

L.ECC : This field is used to store error correcting code in Mode 1. In Mode 2 this field stores additional user data.

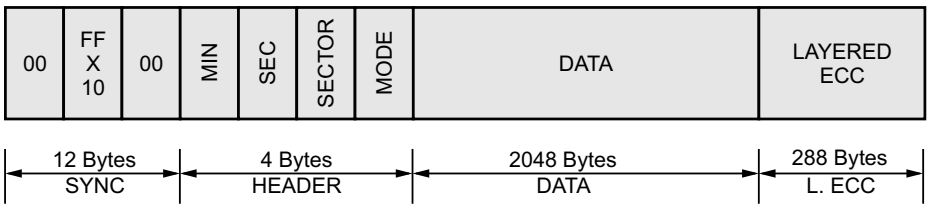


Fig. 4.6.25 Block format for CD-ROM

Advantages of CD-ROM

1. Provides high capacity read-only storage memory.
2. The CD-ROM together with information stored on it can be mass replicated quite economically.
3. It is highly reliable and efficient information storage system.
4. CD-ROM is removable, light in weight and can easily be carried from one computer to another.

Disadvantages of CD-ROM

1. It is read-only memory and can not be updated.
2. Its access time is much higher than the magnetic disk drive.
3. It needs careful handling. Because dust, finger prints or crashes on the reading surface may affect the reproduction of data.

4.6.5.2 WORM

In applications where one or a small number of copies of a set of data is needed, the Write-Once Read-Many (WORM) CD is used. The WORM storage provides one-time writing but unlimited reading of data. Data cannot be overwritten or erased but can be

updated by writing new information into a file at another location on the disk. The new file is then linked to the original file through software.

The WORM disk can be subsequently written once with a laser beam of modest intensity. This requires somewhat more expensive disk controller than for CD-ROM. The WORM uses Constant Angular Velocity (CAV) method to provide more rapid access sacrifying its storage capacity. During preparation of WORM disk a high-power laser is used to produce a series of blisters on the disk. The disk is then placed in a WORM drive to burst the prerecorded blisters with a low powered laser. During a read operation, a laser in the WORM drive illuminates the disk's surface. Since the brust blisters provide higher contrast than the surrounding area, there are easily recognized by simple circuitry.

WORM optical disks are used to handle data that must be occasionally updated and changed. It provides a permanent storage of large volumes of user data.

4.6.5.3 Erasable Optical Disk

Erasable optical storage media is an emerging area where improvements are continually being made. Erasable optical media are used in applications where stored data require frequent changes. Here, magneto-optic technology is used to produce read/write memory. In this technique, the energy of a laser beam is used together with a magnetic field to record and erase information. In magneto-optic technology the write laser heats material in the presence of a magnetic bias field. The heated bit position (a spot on the medium) take on the magnetic polarization of the field and retain it after they cool. As this polarization process does not cause a physical change in the disk, the process can be repeated many times. To erase bits, the bias field is reversed, and all bits are heated by the laser. To read data, the direction of magnetism can be detected by polarized laser light. The polarization of the light striking the written bit positions will be opposite that striking the rest of the media.

Advantages of Erasable Optical Disk

1. Erasable optical disk can be erased and rewritten and thus can be used as a true secondary storage.
2. Provides high storage capacity about 650 Mbytes of data on 5.25 inch disk.
3. It is portable and can easily be carried from one computer to another.
4. It is highly reliable and has longer life.

Disadvantages of Erasable Optical Disk

1. It uses constant angular velocity method, therefore lot of storage space is wasted in the outer tracks.

2. Over writing data on magneto-optic media is slower than for magnetic media, since one revolution is required to erase a bit and a second is required to write back to that location.

4.6.5.4 DVD Technology

DVD technology offers many improvements over earlier electronic resources. DVD disks are more durable and portable than laser disks and more resistant to scratches than CDs. Videotapes wear out, but DVD disks are read by lasers and are not damaged with repeated use. Unlike magnetic media such as floppy disks and videotapes, DVD disks are optical media and cannot be damaged by magnet fields. They are also less sensitive to extreme temperature.

DVD technology also provides higher resolution than videotape, laser disk and CD-or Internet based video.

The main differences between the CD and DVD are summarized in the following Table 4.6.5.

Feature	DVD	CD-ROM
Substrate diameter/thickness (mm)	120/1.2	120/1.2
Sides	1 or 2	1
Layers per side	1 or 2	1
Capacity (GB)	4.7, 8.54, 9.4, or 17	~ 0.7
Track pitch (microns)	0.74	1.6
Min pit length (microns)	0.4 - 0.44	0.83
Linear velocity used for scan (m/s)	3.5 - 3.84	1.3
Laser wavelength (nm)	635 or 650	780
Numerical aperture	0.6	0.45
Modulation	8 to 16	EFM (8 to 17)
Error Correction Code (ECC)	RSPC	CIRC
Durability and dust/scratch	Same as that of CD	High

Table 4.6.5

DVD Formats

DVDs are available in different formats. The Table 4.6.6 summarizes a range of DVD formats and their primary purpose.

DVD format	Description	Educational uses
DVD-Video	Digital video through which users can navigate interactively. Multiple audio tracks allow for different languages and commentaries.	Often simply referred to as "DVD". Quickly replacing VHS tapes for viewing movies at home. Many education at videos are now available on DVD.
DVD-ROM	Digital video, multimedia and instructional content that can be accessed in non-linear fashion by using a computer.	Similar to DVD-Video, but can also hold computer files. Likely to one day replace CD-ROMs.
DVD-RAM	Format that allows re-recording and random access, making it suitable as a peripheral storage device.	Can be used like an additional hard drive for your computer. Not widely used in schools.
DVD-R	Format that allows you to burn (copy files onto) a DVD; cannot be re-recorded.	Use to save data to share with others. Requires special software (such as Appels DVD) to create your own DVD-videos. One advantage to DVD-R is that they can't be overwritten, so there isn't the chance that someone will delete of add information on your DVD.
DVD-RW	Similar to DVD-R, but can be re-recorded.	These discs are re-usable, but are more costly to purchase than DVD-R discs.
DVD-Audio	Format for storing only audio files on DVD.	Likely to one day replace music CDs.

Table 4.6.6

DVD Capacity

The DVD can be of different capacities. As we know, a traditional CD-ROM can contain upto 650 MB of data or 74 minutes of audio. For DVD the capacity depends on three factors :

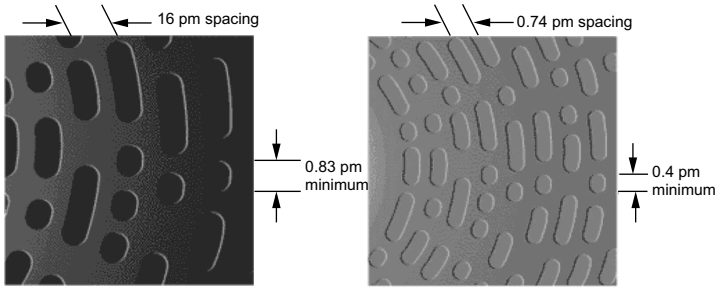


Fig. 4.6.26

1. Both on CD and DVD, data are represented by a series of micro-cuts on a spiral starting from the centre of the disk. In a DVD however, the micro-cuts are smaller and the spiral is tighter than in a CD allowing an higher density of data on the same area.
2. The DVD can have two layers, one above the other, where the data are contained. The first layer is semitransparent and the head laser is calibrated to read it without problems. To read the second layer, the focusing of the laser is changed so that it passes through the first layer and reads the data on the second one. In this way there can be a double amount of data.
3. Each disk can be written on both sides (and each side can have two layers), so the capacity is four times that of the first example; of course you need to turn the disk manually to read the second side.

In detail, following are the possible capacities :

Layers number for each side	Registered sides	Capacity in Gigabyte	Hours
1	1	4.7	2
2	1	8.5	4
1	2	8.7	4.5
2	2	17	8

Features of DVD

The main features of the DVD **formats** can be summarized as follows :

- Backwards compatibility with current CD media (at least the newest models of DVD drives).
- Physical dimensions identical to compact disk with total thickness equal to 1.2 mm, but with capacity at least 7 times larger than that of CD.
- Capacities of 4.7 GB, 8.54 GB, 9.4 GB, and 17.08 GB, depending on the disk structure.
- Single-layer/dual-layer and single/double sided options.
- DVD replication process is similar to that used for compact disks.
- A disk-based format means fast random access like in hard drives and CDs and unlike tapes.
- Designed from the outset for video, audio and multimedia. Meets the requirement for 133 minutes of high quality video on one side of a disk.
- DVD-ROM for enhanced multimedia and games applications.

- DVD-Video for full length high quality movies on one disk.
- DVD-Audio for higher quality music, surround sound and optional video, graphics and other features.
- All formats use a common file system.

DVD Applications

Despite the success of the compact disk there has been a clear need for a higher capacity format to meet additional application requirements.

- **DVD-Video**, which was launched in 1997 in the USA, has become the most successful of all the DVD formats, as it has proved to be an ideal vehicle of distributing video content from the movie industry. It can store a full-length movie of high quality video with surround sound audio on a disk the same size as a CD.
- **DVD-ROM** is beginning to replace the CD-ROM and provide a new high capacity disk format for the computer industry. New PCs are now provided with DVD drives instead of CD drives. The entertainment industry has developed new games consoles (eg Sony's PS2 and Microsoft's X-Box) which incorporate DVD-ROM drives for more sophisticated and realistic games applications.
- **DVD-Audio**, which was launched in 2000, is slowly gathering momentum to become the format for very high quality, surrounded sound music, offering the music industry new revenue opportunities.
- **Recordable** formats such as DVD-RAM, DVD-RW and DVD-R are now being extensively used in PCs for computer backup and short runs of DVDs and in standalone products such as video recorders and camcorders.

DVD technology offers an optical disk with a much larger capacity than the compact disk and is available as a family of pre-recorded, recordable and re-writable formats to meet the requirements of the industries and applications mentioned above.

Review Questions

1. *Explain the organization of magnetic disk in detail.*	**May-07, Marks 6**
2. *Write note on RAID disk arrays.*	**Dec.-07, Marks 16**
3. *Describe the working principle of a typical magnetic disk.*	**Dec.-08, Marks 6**
4. *Describe the working principles of optical disks in detail.*	**May-09, Marks 10**
5. *Write a short note on secondary storage devices in computer system.*	**May-09, Marks 16**
6. *Explain various types of optical memory in detail.*	**May-10, Marks 16**
7. *Explain the organization of magnetic disk in detail.*	**Dec.-10, Marks 16**
8. *Describe in detail magnetic type memories and disk memories.*	**Dec.-11, Marks 16**

4.7 Cache Memory

May-06, 07, 08, 09, 12, Dec.-03, 06, 07, 08, 11

In a computer system, the program which is to be executed is loaded in the main memory (DRAM). Processor then fetches the code and data from the main memory to execute the program. The DRAMs which form the main memory are slower devices. So it is necessary to insert wait states in memory read/write cycles. This reduces the speed of execution. To speed up the process, high speed memories such as SRAMs must be used. But considering the cost and space required for SRAMs, it is not desirable to use SRAMs to form the main memory. The solution for this problem is come out with the fact that most of the microcomputer programs work with only small sections of code and data at a particular time. In the memory system, small section of SRAM is added along with main memory, referred to as **cache memory**. The program which is to be executed is loaded in the main memory, but the part of program (code) and data that work at a particular time is usually accessed from the cache memory. This is accomplished by loading the active part of code and data from main memory to cache memory. The cache controller looks after this swapping between main memory and cache memory with the help of DMA controller. If processor finds that the addressed code or data is not available in cache, then processor accesses that code or data from the main memory (DRAM). The percentage of accesses where the processor finds the code or data word it needs in the cache memory is called the **hit rate**. The hit rate is normally greater than 90 percent.

$$\text{Hit rate} = \frac{\text{Number of hits}}{\text{Number of read/write bus cycles}} \times 100\ \%$$

Example 4.7.1 *The application program in a computer system with cache uses 1400 instruction acquisition bus cycle from cache memory and 100 from main memory. What is the hit rate ? If the cache memory operates with zero wait state and the main memory bus cycles use three wait states, what is the average number of wait states experienced during the program execution ?*

Solution :

$$\text{Hit rate} = \frac{1400}{1400+100} \times 100$$

$$= 93.3333\ \%$$

$$\text{Total wait states} = 1400 \times 0 + 100 \times 3$$

$$= 300$$

$$\text{Average wait states} = \frac{\text{Total wait states}}{\text{Number of memory bus cycles}} = \frac{300}{1500}$$

$$= 0.2$$

4.7.1 Cache Memory System

A cache memory system includes a small amount of fast memory (SRAM) and a large amount of slow memory (DRAM). This system is configured to simulate a large amount of fast memory. Fig. 4.7.1 shows a cache memory system.

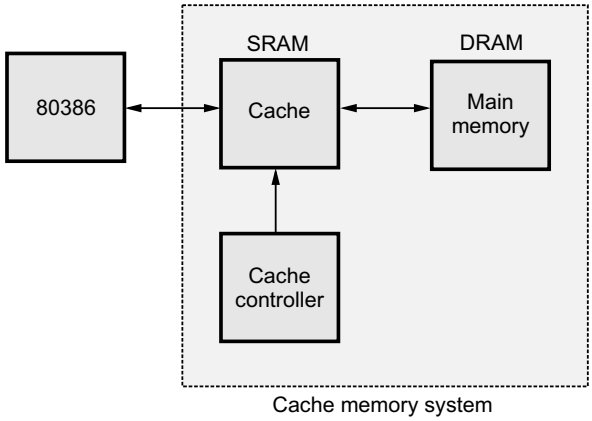


Fig. 4.7.1 Cache memory system

It consists of following sections :

- **Cache :** This block consists of Static RAM (SRAM).
- **Main memory :** This block consists of Dynamic RAM (DRAM).
- **Cache controller :** This block implements the cache logic.

It decides which block of memory should be moved in or out of the cache block and into or out of main memory, based on the requirements.

Two most commonly used system organizations for cache memory are :

- Look-aside and
- Look-through

Look-aside System Organization

The Fig. 4.7.2 shows look-aside system of cache organization. Here, the cache and the main memory are directly connected to the system bus. In this system, the CPU initiates a memory access by placing a physical address on the memory address bus at the start of read or write cycle. The cache memory M_1 immediately compares physical address to the tag addresses currently residing in its tag memory. If a match is found, i.e., in case of cache hit, the access is completed by a read or write operation executed in the cache. The main memory is not involved in the process of read or write. If match is not found, i.e., in case of cache miss, the desired access is completed by a read or write operation directed to M_2. In response to cache miss, a block of data that includes the target address is transferred from M_2 to M_1. The system bus is used for this transfer and hence it is unavailable for other uses like I/O operations.

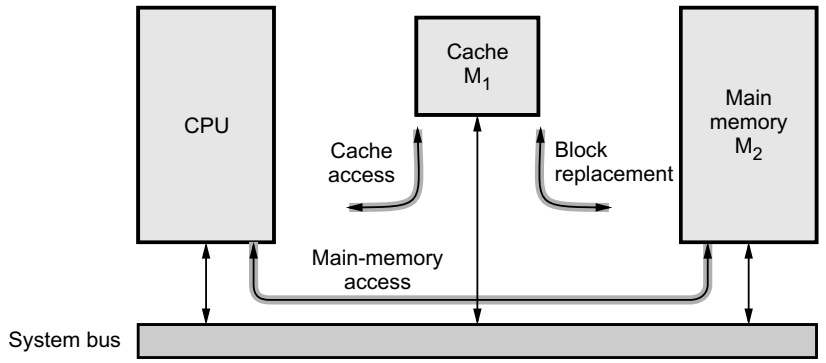


Fig. 4.7.2 Look-aside cache system organization

Look-through System Organization

The Fig. 4.7.3 shows look-through system of cache organization. Here, the CPU communicates with cache via a separate (local) bus which is isolated from the main system bus. Thus during cache accesses, the system bus is available for use by other units, such as I/O controllers, to communicate with main memory. Unlike the look-aside system, look-through cache system does not automatically send all memory requests to main memory; it does so only after a cache miss. A look-through cache systems use wider local bus to link M_1 and M_2, thus speeding up cache-main-memory transfers (block transfers).

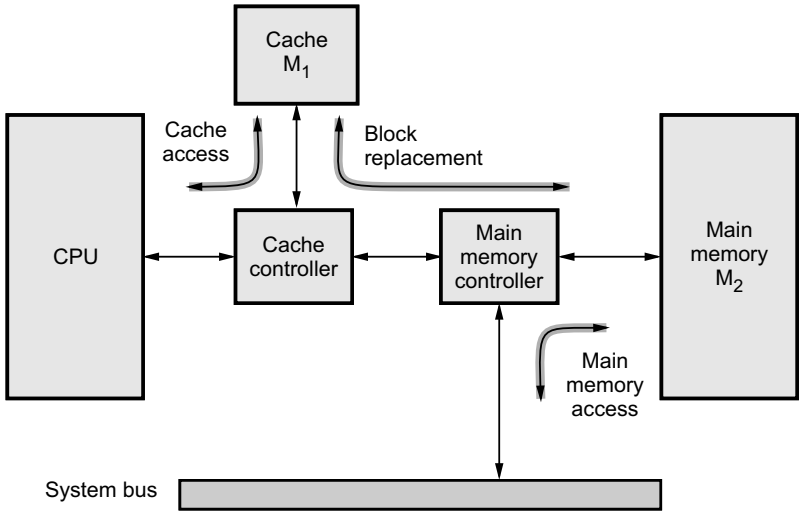


Fig. 4.7.3 Look-through cache system organization

Look-through cache system is faster; however, it has following disadvantages :

- It is complex.
- It is costly.
- It takes longer time for M_2 to respond to the CPU when a cache miss occurs.

Cache Read Operation

The Fig. 4.7.4 shows the small cache system. Here each cache block is 4 bytes and each memory address is 10-bit long. Due to this 8 high-order bits form the tag or block address and the 2 low-order bits define a displacement address within the block. When a block is assigned to the cache data memory, its tag is also placed in the cache tag memory. During read operation, the 8 high-order bits of an address are compared with stored tags in the cache tag memory to find match (cache hit). The stored tag pinpoints the corresponding block in cache data memory and the 2-bit displacement is used to read the target word.

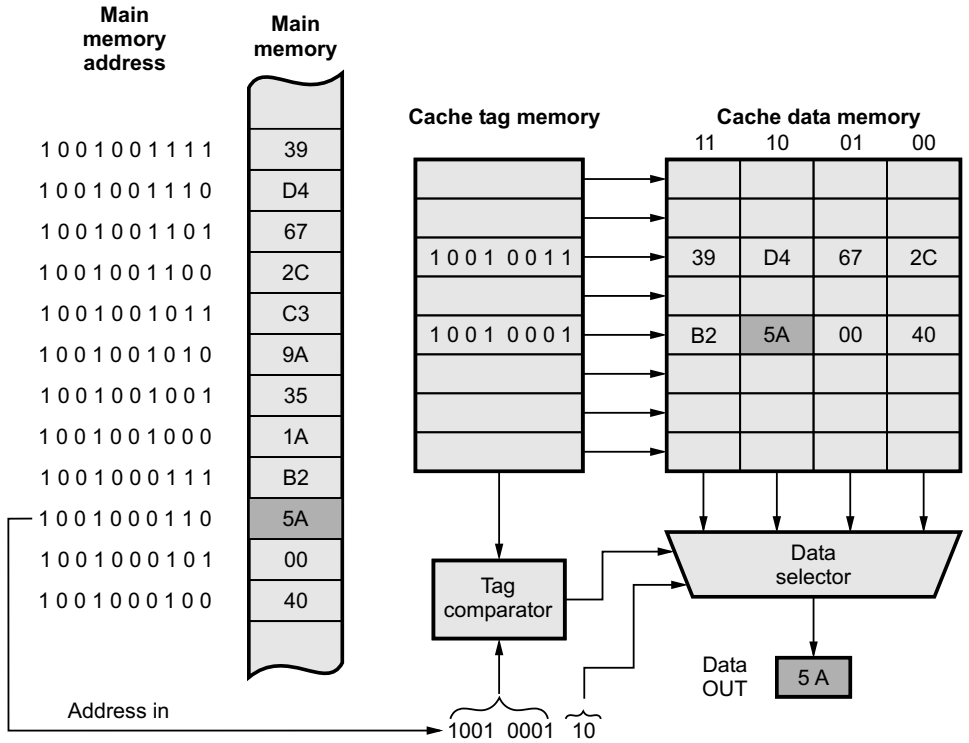


Fig. 4.7.4 Execution of cache read operation

Cache Write Operation

The Fig. 4.7.5 shows execution of cache write operation. It uses same addressing technique as in case of read operation. When cache hit occurs, the new data, in this case E6, is stored at the location pointed by address in the cache data memory, thereby overwriting the old data 5A. Now data in the cache data memory and data in the main memory for given address is different. This causes cache consistency problem. This problem is solved using cache updating systems discussed in section 4.7.5.

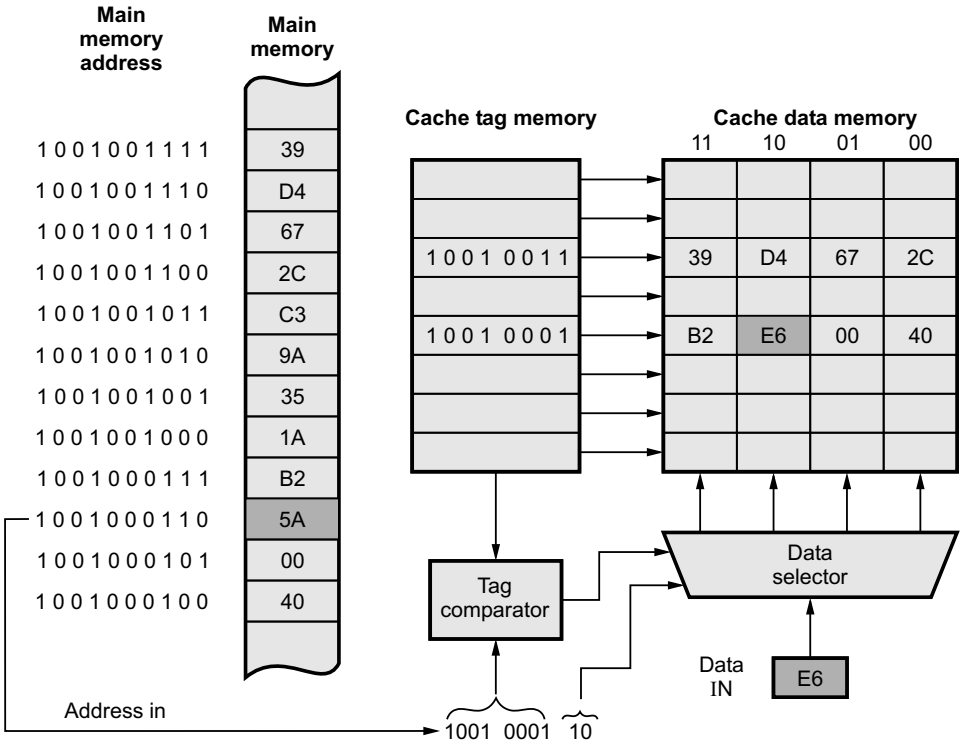


Fig. 4.7.5 Execution of cache write operation

4.7.1.1 Program Locality

In cache memory system, prediction of memory location for the next access is essential. This is possible because computer systems usually access memory from the consecutive locations. This prediction of next memory address from the current memory address is known as **program locality**. Program locality enables cache controller to get a block of memory instead of getting just a single address.

The principle of program locality may not work properly when program executes JUMP and CALL instructions. In case of these instructions, program code is not in sequence.

4.7.1.2 Locality of Reference

We know that program may contain a simple loop, nested loops or a few procedures that repeatedly call each other. The point is that many instructions in localized area of the program are executed repeatedly during some time period and the remainder of the program is accessed relatively infrequently. This is referred to as **locality of reference**. It manifests itself in two ways : **temporal** and **spatial**. The temporal means that a recently

executed instruction is likely to be executed again very soon. The spatial means that instructions stored near by to the recently executed instruction are also likely to be executed soon.

The temporal aspect of the locality of reference suggests that whenever an instruction or data is first needed, it should be brought into the cache and it should remain there until it is needed again. The spatial aspect sugg ests that instead of bringing just one instruction or data from the main memory to the cache, it is wise to bring several instructions and data items that reside at adjacent address as well. We use the term block to refer to a set of contiguous addresses of some size.

4.7.1.3 Block Fetch

Block fetch technique is used to increase the hit rate of cache. A block fetch can retrieve the data located before the requested byte (look behind) or data located after the requested byte (look ahead) or both. When CPU needs to access any byte from the block, entire block that contains the needed byte is copied from main memory into cache.

The size of the block is one of the most important parameters in the design of a cache memory system. The following section describes the pros and cons of small block size and large block size.

PROS and CONS :

1. If the block size is too small, the look-ahead and look-behind are reduced and therefore the hit rate is reduced.
2. Larger blocks reduce the number of blocks that fit into a cache. As the number of blocks decrease, block rewrites from main memory becomes more likely.
3. Due to large size of block, the ratio of required data and useless data is less.
4. Bus size between the cache and the main memory increases with block size to accommodate larger data transfers between main memory and the cache, which increases the cost of cache memory system.

4.7.2 Elements of Cache Design

The cache design elements include cache size, mapping function, replacement algorithm write policy, block size and number of caches.

Cache size : The size of the cache should be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone.

Mapping function : The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the

main memory. Thus we have to use mapping functions to relate the main memory blocks and cache blocks. There are two mapping functions commonly used : direct mapping and associative mapping. Detail description of mapping functions is given in section 4.7.3.

Replacement algorithm : When a new block is brought into the cache, one of the existing blocks must be replaced, by a new block.

There are four most common replacement algorithms :

- Least-Recently-Used (LRU)
- First-In-First-Out (FIFO)
- Least-Frequently-Used (LFU)
- Random

Cache design change according to the choice of replacement algorithm. Detail description of replacement algorithm is given in section 4.7.7.

Write policy : It is also known as cache updating policy. In cache system, two copies of the same data can exist at a time, one in cache and one in main memory. If one copy is altered and other is not, two different sets of data become associated with the same address. To prevent this, the cache system has updating systems such as : write through system, buffered write through system and write-back system. The choice of cache write policy also change the design of cache. Detail description of write policy is given in section 4.7.5.

Block size : It should be optimum for cache memory system. Its importance is already discussed in the section 4.7.1.3.

Number of caches : When on-chip cache is insufficient, the secondary cache is used. The cache design changes as number of caches used in the system changes.

4.7.3 Mapping

Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a **mapping function.**

There are two main mapping techniques which decides the cache organization :

1. Direct-mapping technique
2. Associative-mapping technique

The associative-mapping technique is further classified as fully-associative and set- associative techniques.

To discuss these techniques of cache mapping we consider a cache consists of 128 blocks of 16 words each, for a total of 2048(2 K) words and assume that the main memory has 64 K words. This 64 K words of main memory is addressable by a 16-bit address and it can be viewed as 4 K blocks of 16 words each. The group of 128 blocks of 16 words each in main memory form a page.

4.7.3.1 Direct-Mapping

It is the simplest mapping technique. In this technique, each block from the main memory has only one possible location in the cache organization. In this example, the block i of the main memory maps onto block i module 128 of the cache, as shown in Fig. 4.7.6. Therefore, whenever one of the main memory blocks 0, 128, 256, ... is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257, ... are stored in cache block 1 and so on.

To implement such cache system, the address is divided into three fields, as shown in Fig. 4.7.6. The lower order 4-bits select one of the 16 words in a block. This field is known as **word field**. The second field known as **block field** is used to distinguish a

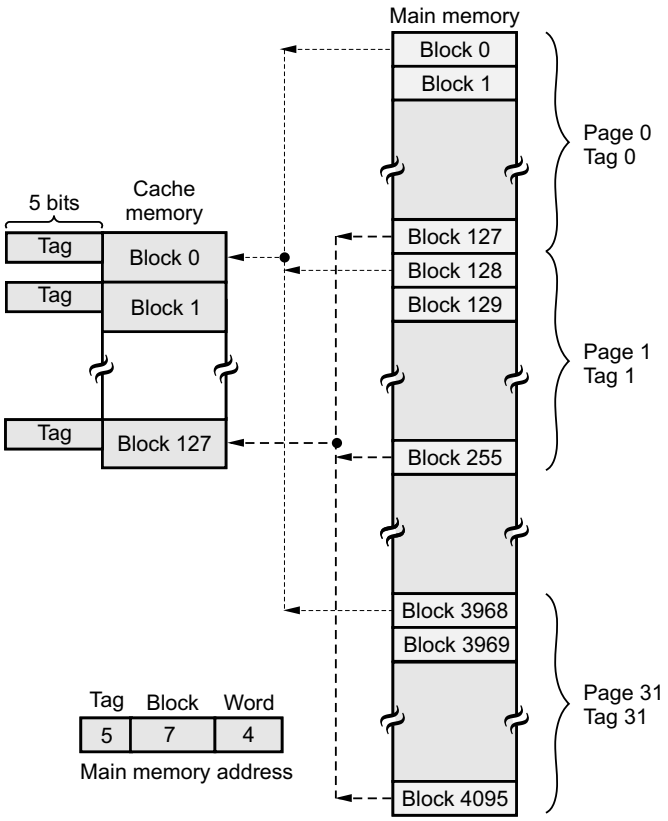


Fig. 4.7.6 Direct-mapped cache

block from other blocks. Its length is 7-bits since 2^7=128. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The third field is a **tag field**. It is used to store the high-order 5-bits of memory address of the block. These 5-bit (tag bits) are used to identify which of the 32 blocks (pages) that are mapped into the cache.

When memory is accessed, the 7-bit cache block field of each address generated by CPU points to a particular block location in the cache. The high-order 5-bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. This means that to determine whether requested word is in the cache, only tag field is necessary to be compared. This needs only one comparison.

The main drawback of direct mapped cache is that if processor needs to access same memory locations from two different pages of the main memory frequently, the controller has to access main memory frequently. Since only one of these locations can be in the cache at a time. For example, if processor want to access memory location 100H from page 0 and then from page 2, the cache controller has to access page 2 of the main memory. Therefore, we can say that direct-mapped cache is easy to implement, however, it is not very flexible.

4.7.3.2 Associative-Mapping (Fully-Associative Mapping)

The Fig. 4.7.7 shows the associative-mapping technique. In this technique, a main memory block can be placed into any cache block position. As there is no fix block, the memory address has only two fields : word and tag. This techniques is also referred to as **fully-associative cache**. The 12-tag bits are required to identify a memory block when it is resident in the cache. The high-order 12-bits of an address received from the CPU are compared to the tag bits of each block of the cache to see if the desired block is present. Once the desired block is present, the 4-bit word is used to identify the necessary word from the cache. This technique gives complete freedom in choosing the cache location in which to place the memory block. Thus, the memory space in the cache can be used more efficiently. A new block that has to be loaded into the cache has to replace (remove) an existing block only if the cache is full. In such situations, it is necessary to use one of the possible replacement algorithm to select the block to be replaced. The replacement algorithms are discussed in the section 4.7.7.

In associative-mapped cache, it is necessary to compare the higher-order bits of address of the main memory with all 128 tag corresponding to each block to determine whether a given block is in the cache. This is the main disadvantage of associative-mapped cache.

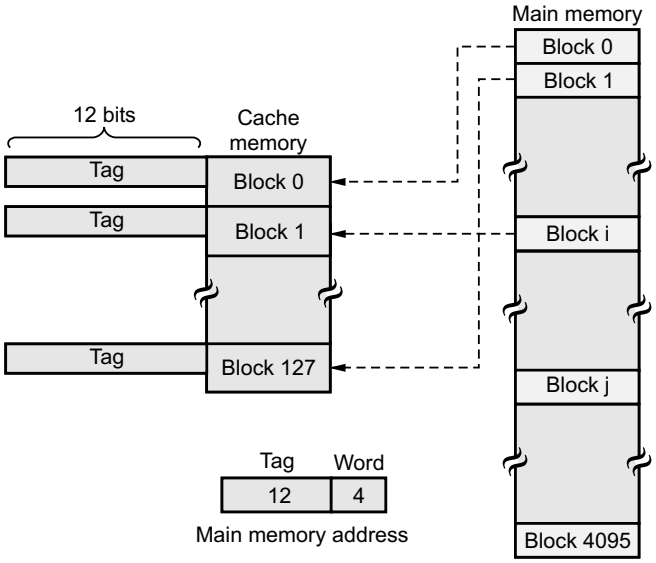


Fig. 4.7.7 Associative-mapped cache

4.7.3.3 Set-Associative Mapping

The set-associative mapping is a combination of both direct and associative mapping. It contains several groups of direct-mapped blocks that operate as several direct-mapped caches in parallel. A block of data from any page in the main memory can go into a particular block location of any direct-mapped cache. Hence the contention problem of the direct-mapped technique is eased by having a few choices for block placement. The required address comparisons depend on the number of direct-mapped caches in the cache system. These comparisons are always less than the comparisons required in the fully-associative mapping.

Fig. 4.7.8 shows two way set-associative cache. Each page in the main memory is organized in such a way that the size of each page is same as the size of one directly mapped cache. It is called two-way set-associative cache because each block from main memory has two choices for block placement. In this technique, block 0, 64, 128,.....4032 of main memory can map into any of the two (block 0) blocks of set 0, block 1, 65, 129,...., 4033 of main memory can map into any of the two (block 1) blocks of set 1 and so on.

As there are two choices, it is necessary to compare address of memory with the tag bits of corresponding two block locations of particular set. Thus for two-way set-associative cache, we require two comparisons to determine whether a given block is in the cache. Since there are two direct-mapped caches, any two bytes having same offset from different pages can be in the cache at a time. This improves the hit rate of the cache system.

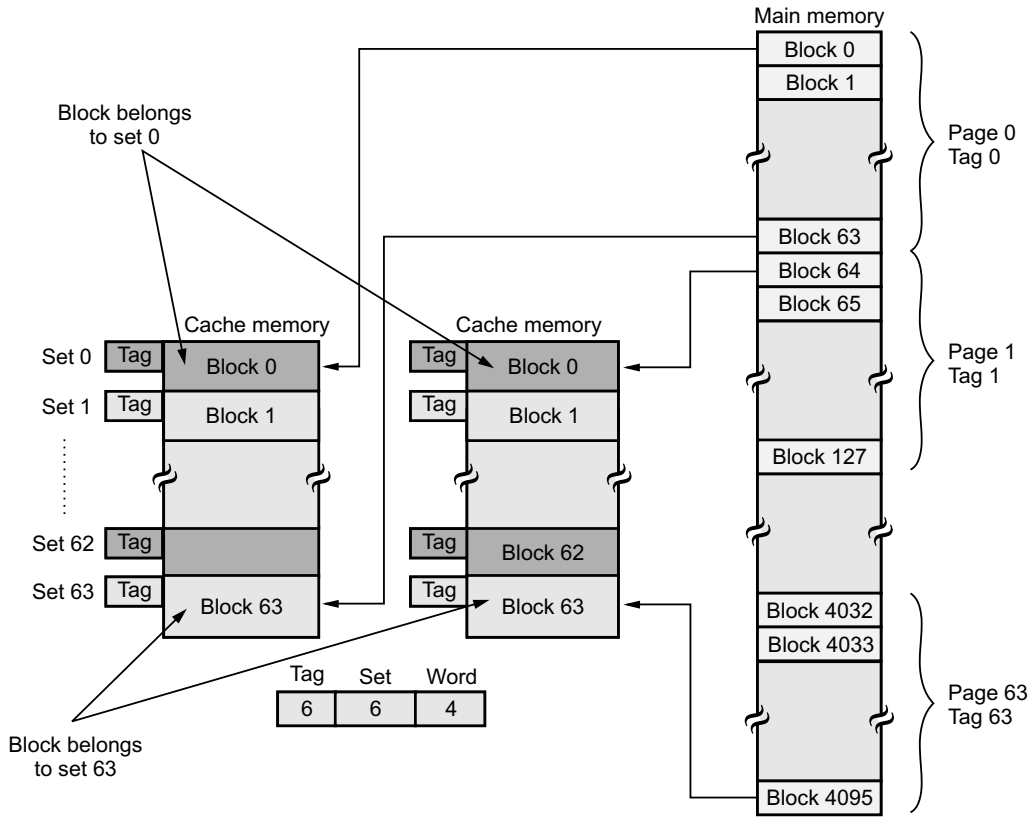


Fig. 4.7.8 Two-way set-associative cache

To implement set-associative cache system, the address is divided into three fields, as shown in Fig. 4.7.8. The 4-bit word field selects one of the 16 words in a block. The set field needs 6-bits to determine the desired block from 64 sets. However, there are now 64 pages. To identify a block belongs to a particular page from 64 pages, six tag bits are required.

Example 4.7.2 *Consider a cache consisting of 256 blocks of 16 words each, for a total of 4096 (4 K) words, and assume that the main memory is addressable by a 16-bit address and it consists of 4 K blocks. How many bits are there in each of the TAG, BLOCK/SET and word fields for different mapping techniques ?*

Solution : We know that memory address is divided into three fields. We will now find the exact bits required for each field in different mapping techniques.

a) Direct-mapping

Word bits : We know that each block consists of 16 words. Therefore, to identify each word we must have (2^4 = 16) four bit reserved for it.

Block bits : The cache memory consists of 256 blocks and using direct-mapped technique, block k of the main memory maps onto block k modulo 256 of the cache. It has one to one correspondence and requires unique address for each block. To address 128 block we require (2^8 = 256) eight bits.

Tag bits : The remaining 4 (16 – 4 – 8) address bits are tag bits which stores the higher address of the main memory.

The main memory address for direct-mapping technique is divided as shown below :

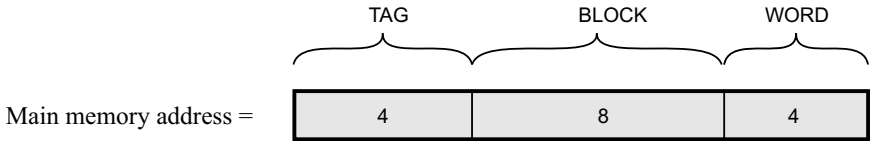


Fig. 4.7.9 (a)

b) Associative-mapping

In the associative-mapping technique, each block in the main memory is identified by the tag bits and an address received from the CPU is compared with the tag bits of each block of the cache to see if the desired block is present. Therefore, this type of technique does not have block bits, but all remaining bits (except word bits) are reserved as tag bits.

Word bits : The word length will remain same i.e. 4 bits.

Block bits : 0

Tag bits : To address each block in the main memory (2^{12} = 4096) 12 bits are required and therefore, there are 12 tag bits.

The main memory address for direct mapping technique is divided as shown below :

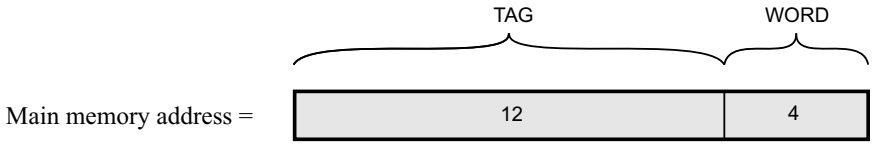


Fig. 4.7.9 (b)

c) Set-associative mapping

Let us assume that there is a 2-way set-associative mapping. Here, cache memory is mapped with two blocks per set. The set field of the address determines which set of the cache might contain the desired block.

Word bits : The word length will remain same i.e. 4 bits

Set bits : There are 128 sets (256/2). To identify each set (2^7 = 128) seven bits are required.

Tag bits : The remaining 5 (16 – 4 – 7) address bits are the tag bits which stores higher address of the main memory.

The main memory address for 2-way set associative mapping technique is divided as shown below :

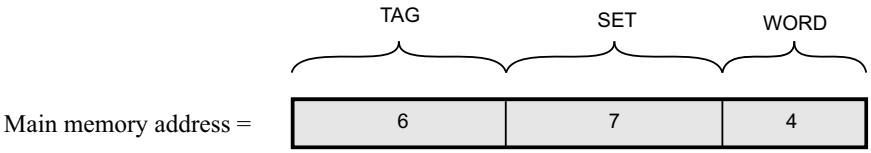


Fig. 4.7.10

Example 4.7.3 *A block set-associative cache consists of a total of 64 blocks divided into four-block sets. The main memory contain 4096 blocks, each consists of 128 words.*

a) How many bits are there in a main memory address ?

b) How many bits are there in each of the TAG, SET and WORD fields ?

c) What is the size of cache memory ?

Solution : a) The main memory contains 4096 blocks and each block consists of 128 words.

$$\therefore \quad \text{Total words in MM} = 4096 \times 128$$

$$= 524288$$

To address 524288 words, we require (2^{19} = 524288) 19 bits.

b) Word bits : There are 128 words in each block. Therefore, to identify each word (2^7 = 128) 7 bits are required.

Set bits : There are 64 blocks and each set consists of 4 blocks. Therefore, there are 16 (64/4) sets. To identify each set (2^4 = 16) four bits are required.

Tag bits : We know that total address bits are 19 bits. Therefore tag bits are 8 (19 – 7 – 4).

The main memory address for 4-way set-associative mapping is divided as shown below :

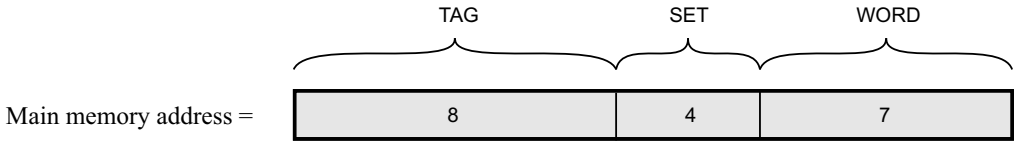


Fig. 4.7.11

c) The cache memory has 64 blocks of 128 words each. Therefore,

Total words in cache memory = $64 \times 128 = 8192$

Example 4.7.4 *A block set-associative cache consists of 64 blocks divided into 4 block sets. The main memory contains 4096 blocks, each consists of 128 words of 16 bits length :*

i) How many bits are there in main memory ?

ii) How many bits are there in each of the TAG, SET and WORD fields ?

Solution : i) Number of bits in main memory :

= Number of blocks × Number of words per block × Number of bits per word

= 4096 × 128 × 16

= 8388608 bits

ii) Number of bits in word field : There are 128 words in each block. Therefore, to identify each word ($2^7 = 128$) 7 bits are required.

iii) Number of T bits in set field : There are 64 blocks and each set consists of 4 blocks.

Therefore, there are 16 (64/4) sets. To identify each set ($2^4 = 16$) four bits are required.

iv) Number of bits in tag field : The total words in the memory are :

$$4096 \times 128 = 524288.$$

To address these words we require ($2^{19} = 524288$) 19 address lines. Therefore, tag bits are eight (19 – 7 – 4).

Example 4.7.5 *A digital computer has a memory unit of 64 K × 16 and a cache memory of 1 K words. The cache uses direct mapping with a block size of four words. How many bits there in the tag index, block and word field of the address format ? How many block can the cache accommodate.* **May-07, Marks 10**

Solution :

Word bits : Number of word bits = $\log_2 4 = \log_2 2^2 =$ **2-bits**

Block bits : Number of block $= \dfrac{\text{Cache size}}{\text{Words in each block}}$

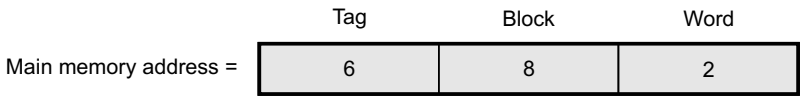
$$= \frac{1\text{ K}}{4} = \mathbf{256}$$

$\therefore$ Number of block bits = $\log_2 256 = \log_2 2^8 =$ **8 bits**

Tag bits : Number of bits to address main memory

$$= \log_2 64\text{ K} = \log_2 2^{16} = \mathbf{16\ bits}$$

$\therefore$ Number of Tag bits = 16 – 8 – 2 = **6 bits**



The cache can accommodate 256 blocks.

Example 4.7.6 *A two way set associative cache memory uses block of four words. The cache can accommodate a total of 2048 words from main memory. The main memory size is 128 K × 32.*

i) How many bits are there in the tag index, block and word field of address format ?

ii) What is size of cache memory ?

Solution :

Number of bits in main memory address = $\log_2 128\text{ K} = \log 2^{17} =$ **17 bits**

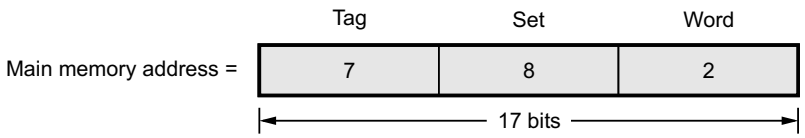
Number of blocks in the cache memory = $\dfrac{2048}{4} =$ **512 Blocks**

Number of sets in the cache memory = $\dfrac{512}{2} =$ **256 sets**

Number of bits in set field = $\log_2 256 = \log_2 2^8 =$ **8 bits**

Number of bits in word field = $\log_2 4 = \log_2 2^2 =$ **2 bits**

Number of bits in tag field = 17 – 8 – 2 = **7 bits**



Example 4.7.7 *An eight-way set-associative cache consists of a total of 256 blocks. The main memory, contains 8192 blocks each consisting of 128 words.*

a) How many bits are there in the main memory address ?

b) How many bits are there in the TAG, SET and WORD fields ? **May-06, Marks 2**

Solution : a) The main memory contains 256 blocks, and each block consists of 128 words.

$\therefore$ Total words in MM = $8192 \times 128 = 1048576$

To address 32768 words we required ($2^{20} = 1048576$) 20 bits.

b) Word bits : There are 128 words in each block. Therefore, to identify each word ($2^7 = 128$) 7 bits are required.

Set bits : There are 256 blocks. Since it is a 8-way cache number of set are (256/8) 16. To identify each set ($2^4 = 16$) four bits are required.

Tag bits : The remaining 9 (20 – 4 – 7) address bits are the tag bits which stores higher address of the main memory.

Example 4.7.8 *A computer system has a main memory consisting of 16 M words. It also has a 32K-word cache organized in the block-set-associative manner, with 4 blocks per set and 128 words per block.*

Calculate the number of bits in each of the TAG, SET and WORD fields of the main memory address format.

How will the main memory address look like for a fully associative mapped cache ?

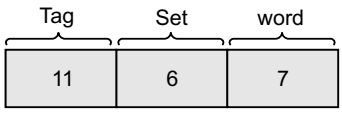
May-07, Marks 8

Solution : Word bits : Each block consists of 128 words. Therefore, to identify each word we must have ($2^7 = 128$) 7 bits reserved for it.

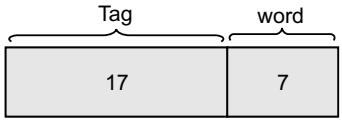
Set bits : There are 64 sets (256/4). To identify each set ($2^6 = 64$) six bits are required.

Tag bits : Since the main memory has 16 M words, we require 24 bit address. Therefore the tag bits = 24 – 6 – 7 = 11 bits.

Main memory address format =



Main memory address for fully associative mapped cache =



4.7.4 Comparison between Mapping Techniques

Sr. No.	Direct-mapping	Associative-mapping	Set-associative-mapping
1.	Each block from the main memory has only one possible location in the cache.	A block of data from main memory can be placed into any cache block position.	A block of data from main memory can go into a particular block location of any direct-mapped cache.
2.	Needs only one comparison.	Needs comparison with all tag bits.	Needs number of comparisons equal to number of blocks per set.
3.	Cache hit ratio decreases if processor needs to access same memory location from two different pages of the main memory frequently.	Cache hit ratio has no effect if processor needs to access same memory location from two different pages of the main memory frequently.	The effect of reduction in cache hit ratio in case of frequent access to the two different pages of the main memory is reduced.
4.	Main memory address is divided into three fields : TAG, BLOCK and WORD.	Main memory address is divided into two fields : TAG and WORD.	Main memory address is divided into three fields : TAG, SET and WORD.
5.	Searching time is less.	Searching time is more.	Searching time increases with number of blocks per set.

Table 4.7.1 Comparison between mapping techniques

4.7.5 Cache Write / Updating

In a cache system, two copies of same data can exist at a time, one in cache and one in main memory. If one copy is altered and other is not, two different sets of data become associated with the same address. To prevent this, the cache system has updating systems such as : Write through system, buffered write through system and write-back system.

4.7.5.1 Write through System

In write through updating system, the cache controller copies data to the main memory immediately after it is written to the cache. Due to this, main memory always contains a valid data and any block in the cache can be overwritten immediately without data loss.

The write through is a simple approach. This approach requires time to write data in main memory with increase in bus traffic. This in effect reduces the system performance.

4.7.5.2 Buffered Write through System

In buffered write through system, the processor can start a new cycle before the write cycle to the main memory is completed. This means that the write accesses to the main memory are buffered. In such systems, read access which is a "cache hit" can be performed simultaneously when main memory is updated. However, two consecutive write operations to the main memory or read operation with cache "miss" require the processor to wait.

4.7.5.3 Write-Back System

In a write-back system, the alter bit in the tag field is used to keep information of the new data. Cache controller checks this bit before overwriting any block in the cache. If it is set, the controller copies the block to main memory before loading new data into the cache.

Due to one time write operation, number of write cycles are reduced in write-back system. But this system has following disadvantages.

- Write-back cache controller logic is more complex.
- It is necessary that, all altered blocks must be written to the main memory before another device can access these blocks in main memory.
- In case of power failure, the data in the cache memory is lost, so there is no way to tell which locations of the main memory contain old data. Therefore, the main memory as well as cache must be considered volatile and provisions must be made to save the data in the cache.

4.7.6 Cache Coherency

Cache updating systems eliminates data inconsistency in the main memory caused by cache write operations. However, in multiprocessor systems, another bus master can take over control of the system bus. This bus master could write data into a main memory blocks which are already held in the cache of another processor. When this happens, the data in the cache no longer match those held in main memory creating inconsistency. The 80386 supports four different approaches to prevent data inconsistency that is to protect cache coherency :

1. Bus watching (snooping) 2. Hardware transparency
3. Non-cacheable memory 4. Cache flushing.

Bus watching : In bus watching, cache controller invalidates the cache entry, if another master writes to a location in shared memory which also resides in the cache memory. Fig. 4.7.12 shows bus watching.

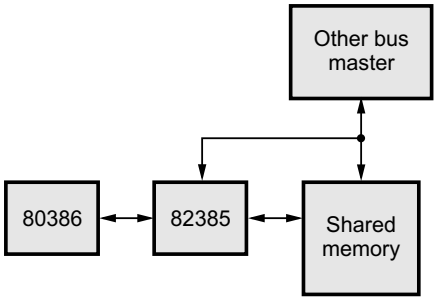


Fig. 4.7.12 Bus watching

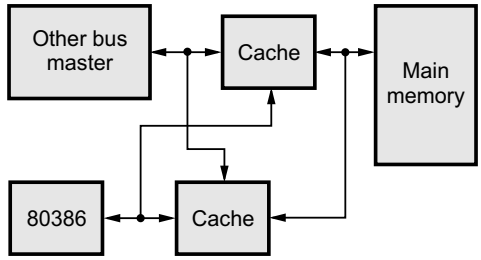


Fig. 4.7.13 Hardware transparency

Hardware transparency : In hardware transparency, accesses of all devices to the main memory are routed through the same cache or by copying all cache writes both to the main memory and to all other caches that share the same memory. Fig. 4.7.13 shows hardware transparent system.

Non-cacheable memory : The 80386DX can partition its main memory into a cacheable and non-cacheable memory. By designing shared memory as non-cacheable memory cache coherency can be maintained, since shared memory is never copied into cache.

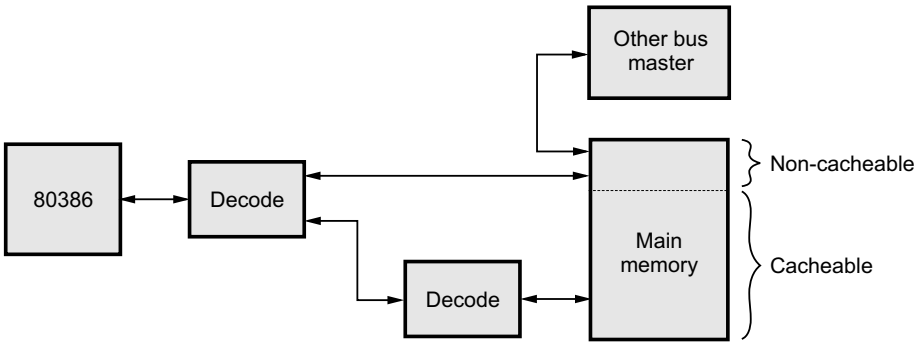


Fig. 4.7.14 Memory

Cache flushing : To avoid data inconsistency, a cache flush writes any altered data to the main memory and caches in the system are flushed before a device writes to shared memory.

4.7.7 Replacement Algorithms

When a new block is brought into the cache, one of the existing blocks must be replaced, by a new block. In case of direct-mapping cache, we know that each block from the main memory has only one possible location in the cache, hence there is no choice. The previous data is replaced by the data from the same memory location from new page of the main memory. But for associative and set-associative techniques, there is a choice of replacing existing block. The choice of replacement of the existing block

should be such that the probability of accessing same block must be very less. The replacement algorithms do the task of selecting the existing block which must be replaced.

There are four most common replacement algorithms :

- Least-Recently-Used (LRU)
- First-In-First-Out (FIFO)
- Least-Frequently-Used (LFU)
- Random

Least-Recently-Used : In this technique, the block in the set which has been in the cache longest with no reference to it, is selected for the replacement. Since we assume that more-recently used memory locations are more likely to be referenced again. This technique can be easily implemented in the two-way set-associative cache organization.

First-In-First-Out : This technique uses same concept that stack implementation uses in the microprocessors. In this technique, the block which is first loaded in the cache amongst the present blocks in the cache is selected for the replacement.

Least-Frequently-Used : In this technique, the block in the set which has the fewest references is selected for the replacement.

Random : Here, there is no specific criteria for replacement of any block. The existing blocks are replaced randomly. Simulation studies have proved that random replacement algorithm provides only slightly inferior performance to algorithms just discussed.

Review Questions

1. *Explain the cache execution of a read operation and its organization.* **Dec.-03, Marks 16**
2. *Discuss the various mapping schemes used in cache design. Compare the schemes in terms of cost and performance.* **May-06, Marks 10**
3. *What do you understand by hit ratio ?* **Dec.-06, Marks 2**
4. *Define locality of reference. What are its types ?* **Dec.-06, Marks 2**
5. *Explain the various mapping functions used for mapping main memory blocks into cache memory.* **Dec.-06, May-12, Marks 8**
6. *Discuss the various mapping techniques used in cache memories.* **May-07, Marks 8**
7. *Write note on set associative mapping of cache.* **Dec.-07**
8. *Describe in detail about the memory mapping between virtual and main memory with an example.* **Dec.-07, Marks 16**
9. *Define the terms : Spatial locality and temporal locality.* **May-08, Marks 2**

10. *Explain various mechanisms of mapping main memory address into cache memory addresses.* **May-08, Marks 10**
11. *Define Locality of reference.* **May-08, Marks 2**
12. *Define cache mapping functions. Explain the methods.* **May-08, Marks 2**
13. *Describe the working principle of RAM, ROM and cache memories. Compare them based on their speed, size and cost.* **Dec.-08, Marks 16**
14. *Explain about direct mapping and set associative mapping in cache memory.* **Dec.-08, Marks 16**
15. *Explain the various cache replacement policies in the cache memory.* **May-09, Marks 16**
16. *Describe cache memory in detail.* **Dec.-11, Marks 16**
17. *Why do we use cache memory ? and explain the different types of mapping functions with the necessary block diagrams.* **May-12, Marks 16**

4.8 Performance Issues

May-06, 07, 10

The overall goal in memory-hierarchy design should be to achieve a performance close to that of the fastest device M_1 and a cost per bit close to that of the cheapest device M_n. The performance of a memory system depends mainly on following factors.

- **Address reference statistics :** It means the order and frequency of the logical address generated by programs that use the memory hierarchy.
- **Access time (t_A) :** The access time (t_A) of each memory level relative to the CPU.
- **Storage capacity (S) :** Storage capacity of each memory level.
- **Block size :** The size of the blocks (pages) transferred between adjacent levels.
- **Allocation algorithm :** The algorithm used to determine the regions of memory to which blocks are transferred by the block replacement process.

For simplicity consider two level memory hierarchy denoted by (M_1, M_2), which can be interpreted as (cache, main memory) or (main memory, secondary memory). Then the average cost per bit of memory is given by,

$$c = \frac{c_1 S_1 + c_2 S_2}{S_1 + S_2} \quad \ldots (4.8.1)$$

where c is the cost per bit of memory.

S is the storage capacity of memory in bits.

We know that, the performance of a two-level memory is often measured in terms of the **hit rate, h**. Let us consider M_1 is the fast memory and M_2 is the slow memory. The number of address references satisfied by M_1 and M_2 are N_1 and N_2 , respectively. Then hit ratio is given by,

$$h = \frac{N_1}{N_1 + N_2} \quad \ldots (4.8.2)$$

The quantity 1 – h is called the **miss rate**. It is desirable to make h as close to one as possible. Let us consider t_{A1} and t_{A2} are the access times of M_1 and M_2, respectively, relative to the CPU. The average time t_A for the CPU to access a word in the two-level memory is given by,

$$t_A = ht_{A1} + (1 - h)\, t_{A2} \quad \ldots (4.8.3)$$

In case of cache miss, the extra time needed to bring the requested word into the cache is called the **miss penalty**. This penalty is ultimately reflected in the time that the processor is stalled because the required instruction or data are not available for execution in fast memory, M_1. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit, i.e. tA_2 in the above given expression. It is also denoted as M (miss penalty). Similarly, if M_1 is the cache memory than tA_1 which is the access time of M_1 is denoted as C and we can write equation (4.8.3) as,

$$t_A = hC + (1 - h)\, M$$

The time required for block transfer is denoted as t_B. Then the time required to access slow memory M_2 can be given by,

$$t_{A2} = t_B + t_{A1} \quad \ldots (4.8.4)$$

Substituting this value of t_{A2} in equation (4.8.3) we get,

$$\begin{aligned} t_A &= ht_A + (1 - h)\, (t_B + t_{A1}) \\ &= ht_A + (1 - h)\, t_B + (1 - h)\, t_{A1} \\ &= ht_A + (1 - h)\, t_B + t_{A1} - ht_{A1} \\ &= t_{A1} + (1 - h)\, t_B \quad \ldots (4.8.5) \end{aligned}$$

In most of the cases $t_{A2} >> t_{A1}$; therefore, $t_{A2} \approx t_B$ and we have,

$$t_A = t_{A1} + (1 - h)\, t_{A2} \quad \ldots (4.8.6)$$

The access time ratio $\mathbf{t_{A2}/t_{A1}}$ of the two levels of memory is denoted as **r** and the access efficiency $\mathbf{t_{A1}/t_A}$ is denoted as **e**.

Therefore, from equation (4.8.3) we have,

$$e = \frac{t_{A1}}{t_A} = \frac{t_{A1}}{ht_{A1} + (1 - h)\, t_{A2}}$$

Dividing numerator and denominator by t_{A1} we get,

$$e = \frac{1}{h+(1-h)\,r} \quad \text{where } r = \frac{t_{A2}}{t_{A1}}$$

$$= \frac{1}{h+r-hr} = \frac{1}{r+(1-r)\,h}$$

Example 4.8.1 *Consider a two-level memory hierarchy M_1 and M_2 and let c_1, c_2 be the costs per type, t_{A1} and t_{A2} be the access times and S_1 and S_2 be the memory capacities for M_1 and M_2 respectively.*

a) Under what conditions will the average cost of the entire memory system approach c_2.

b) What is the effective memory access time t_A of this hierarchy.

c) Express access efficiency e in term of speed ratio and hit ratio.

d) Plot e against h for r = 5 and 50 and comment on its performance.

Solution : a) The average cost of the entire memory system, c will approach c_2 when S_1 is much smaller than S_2 .

b) Memory access $t_A = ht_{A1} + (1 - h)\, t_{A2}$

c) Access efficiency $e = \dfrac{1}{r+(1-r)\,h}$

d) To plot e as a function of h, we tabulate the values of r, e and h.

Hit ratio		**Access efficiency** $e = \dfrac{1}{r+(1-r)\,h}$	
r = 5	**r = 50**	**r = 5**	**r = 50**
0	0	0.2	0.02
0.2	0.2	0.238	0.0248
0.4	0.4	0.294	0.0329
0.6	0.6	0.385	0.0485
0.8	0.8	0.555	0.0926
1.0	1.0	1	1

The Fig. 4.8.1 shows plots e as a function of h for r = 5 and r = 50.

The plot shows that to achieve high efficiency, we must have high hit ratio. For higher access time ratio, the above constraint becomes more severe.

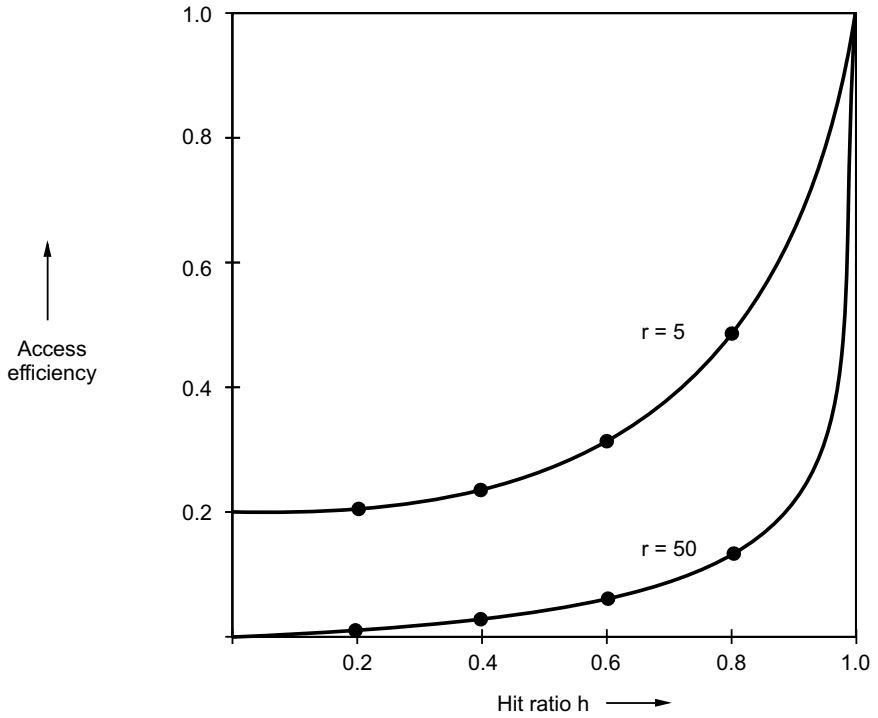


Fig. 4.8.1 Plots e as a function of h for r = 5 and r = 50

Example 4.8.2 *Consider a two-level memory hierarchy of the form (M_1 , M_2) where M_1 is connected directly to the CPU. Determine the average cost per bit c and average access time t_A for the data given below.*

Level (i)	Capacity (S_i)	Cost (c_i)	Access time (t_{Ai})	Hit ratio (H)
M_1 (Cache)	1024	0.1000	10^{-8}	0.9000
M_2 (Main)	2^{16}	0.0100	10^{-6}	-

Solution : $$\text{Average cost (c)} = \frac{c_1 S_1 + c_2 S_2}{S_1 + S_2} = \frac{0.1 \times 1024 + 0.01 \times 2^{16}}{1024 + 2^{16}} = 0.011384615$$

Average access time,

$$t_A = h t_{A1} + (1 - h)\, t_{A2} = 0.9000 \times 10^{-8} + (1 - 0.9000)\, 10^{-6}$$

$$= 1.09 \times 10^{-7}$$

Example 4.8.3 *In a two-level virtual memory, $t_{A1} = 10^{-7}$ s and $t_{A2} = 10^{-2}$ s. What must be the hit ratio h be in order for the access efficiency to be at least 90 percent of its maximum possible value.*

Solution : $t_{A1} = 10^{-7}$ e

$= 90\ \% = 0.9$

$t_{A2} = 10^{-2}$, Hit ratio h = ?

$$e = \frac{1}{r + (1 - r)\, h}$$

and $$r = \frac{t_{A2}}{t_{A1}} = \frac{10^{-2}}{10^{-7}} = 10^5$$

$$0.9 = \frac{1}{10^5 + (1 - 10^5)\, h}$$

$$h = 0.999999$$

Example 4.8.4 *Consider a processor running a program 30 % of the instructions of which require a memory read or write operation if the cache hit ratio is 0.95 for instructions and 0.9 for data. When a cache hit ratio occurs for instruction or for data, only one clock is needed while the cache miss penalty is 17 clocks to read/write on the main memory. Work out the time saved by using the cache, given the total number of instructions executed is 1 million.*

Solution : It is given that 30 % of the instructions require a memory read or write operation, which means that there are 130 memory accesses for every 100 instructions executed. It is also given that cache hit ratio is 0.95 for instructions and 0.90 for data.

Assuming time required for each memory access with no cache is 10 clock cycles, time required for 1 million instructions without cache can be given as,

$$\text{Time without cache (1 million)} = (1 \times 10^6 + 1 \times 10^6 \times 0.3) \times 10$$

$$= 13 \times 10^6 \text{ clock cycles}$$

For 1 million instructions, time required with cache can be given as,

$$\text{Time with cache (1 million)} = 1 \times 10^6\, (0.95 \times 1 + 0.05 \times 17) + (1 \times 10^6 \times 0.3)(0.9 \times 1 + 0.1 \times 17)$$

$$= 1.8 \times 10^6 + 0.78 \times 10^6$$

$$= 2.58 \times 10^6 \text{ clock cycles}$$

$\therefore$ Time saved by using the cache for 1 million instructions is,

$$T_{saved} = 13 \times 10^6 - 2.58 \times 10^6 = 10.42 \times 10^6 \text{ clock cycles}$$

Example 4.8.5 *Calculate the average access time of memory for a computer with cache access time of 100 ns, a main memory access of 1000 ns and a hit ratio is 0.9.*

Solution : From equation (4.8.6) of section 4.8 the average access time is given by,

$$t_A = t_{A1} + (1 - h)\, t_{A2}$$
$$= 100 \text{ ns} + (1 - 0.9) \times 1000 \times 10^{-9} = \mathbf{200 \text{ ns}}$$

Example 4.8.6 *Consider a two level cache with access times of 5 ns and 80 ns respectively. If the hit rates are 92 % and 75 % respectively in the two caches and the memory access time is 250 ns, what is the average access time ?* **May-06, Marks 6**

Solution :

$$t_A = h_1\, t_{A1} + (1 - h_1)\, h_2 t_{A2} + (1 - h_1)\,(1 - h_2)\, t_{A3}$$
$$= 0.92 \times 5 + (1 - 0.92) \times 0.75 \times 80 + (1 - 0.92)\,(1 - 0.75) \times 250$$
$$= 4.6 + 4.8 + 0.4 = \mathbf{9.8 \text{ ns}}$$

Review Questions

1. *List the factors that determine the storage device performance.* **May-07, Marks 2**
2. *Define average memory access time for a computer system with two levels of caches.* **May-07, Marks 2**
3. *How is the average access time of two level memory systems computed ?* **May-10, Marks 2**

4.9 Virtual Memory

May-06, 07, 08, 09, 10, 11, Dec.-06, 07, 08, 10, 12

In most modern computers, the physical main memory is not as large as the address space spanned by an address issued by the processor. Here, the virtual memory technique is used to extend the apparent size of the physical memory. It uses secondary storage such as disks, to extend the apparent size of the physical memory. Let us see how this technique works.

When a program does not completely fit into the main memory, it is divided into segments. The segments which are currently being executed are kept in the main memory and remaining segments are stored in the secondary storage devices, such as a magnetic disk. If an executing program needs a segment which is not currently in the main memory, the required segment is copied from the secondary storage device. When a new segment of a program is to be copied into a main memory, it must replace another segment already in the memory. In modern computers, the operating system moves program and data automatically between the main memory and secondary storage. Techniques that automatically swaps program and data blocks between main memory and secondary storage device are called **virtual memory**. The addresses that processor issues to access either instruction or data are called **virtual** or **logical address**.

The set of such addresses is called **address space.** Thus the address space is the set of addresses generated by programs as they reference instruction and data. These addresses are translated into physical addresses by a combination of hardware and software components. The physical address is an address of main memory. The set of such addresses is called the **memory space.** The memory space consists of the actual main memory locations directly addressable for processing. The address space is allowed to be larger than the memory space in computers with virtual memory. If a virtual address refers to a part of the program or data space that is currently in the main memory, then the contents of the appropriate location in the main memory are accessed immediately. On the other hand, if the referenced address is not in the main memory, its contents must be brought into a suitable location in the main memory before they can be used.

We have seen that, how virtual memory removes the programming burdens of a small, limited amount of main memory. Along with this it also allows efficient and safe sharing of memory among multiple programs. Consider a number of programs running at once on a computer. The total memory required by all the programs may be much larger than the amount of main memory available on the computer, but only a fraction of this memory is actively being used at any point in time. The main memory need to contain only the active portions of the many programs. This allows use to efficiently share the processor as well as the main memory.

4.9.1 Address Mapping using Pages

Let us assume that address space is 8 K and memory space is 4 K. For mapping purpose, address space and memory space are divided into group of equal sizes. Let us assume size is 1 K words. Thus address space is divided into 8 equal groups called **pages** and memory space is divided into 4 equal groups called **blocks.** This is illustrated in Fig. 4.9.1.

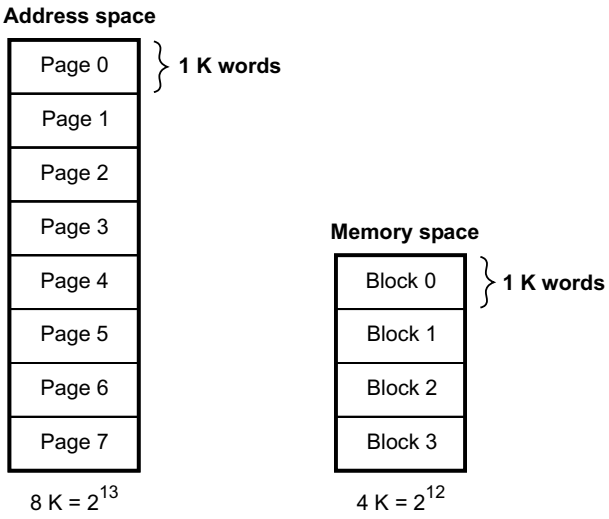


Fig. 4.9.1 Address space and memory space divided into groups of 1 K words

Example 4.9.1 *An address space is specified by 32 bits and corresponding memory space by 24 bits.*

i) How many words are there in the address space ?

ii) How many words are there in the memory space ?

iii) If a page consists of 4 K words, how many pages and blocks are there in the systems.

Solution :

i) Words in the address space = 2^{32} = **4 G words**

ii) Words in the memory space = 2^{24} = **16 M words**

iii) Number of pages $= \frac{\text{Words in address space}}{\text{Words per page}}$

$= \frac{4\text{ G words}}{4\text{ K words}}$ = **1 M pages**

iv) Number of blocks $= \frac{\text{Words in memory space}}{\text{Words per page(block)}}$

$= \frac{16\text{ M words}}{4\text{ K words}}$ = **4 K blocks**

The Fig. 4.9.2 Shows the organization of the memory mapping table in a paged system. The memory page table consists of one word space for each page and word corresponds to particular page is accessed by the page number. In our case, there are 8 pages in the address space so there are 8 words in the memory page table. The three high-order bits of the virtual address specify a page number and it is used as an address for the memory page table. The word in the memory page table gives the block number where that page is stored in the main memory. If the word is empty, i.e., presence bit is 0, the corresponding page is not available in the main memory. The memory page table in the Fig. 4.9.2 shows that page 1, 2, 4 and 6 are now available in the main memory in blocks 2, 0, 1 and 3, respectively. (See Fig. 4.9.2 On next page)

In the mapping process, the contents of the word in the memory page table at the page number address is read out into the main memory address register. The 10 low-order bits of virtual address (line number) are transferred into the 10 low-order bits of the main memory address register forming the (2 + 10) 12-bit address for the main memory.

4.9.2 Page Replacement

If the present bit corresponds to the word in the memory page table at the page number address is 0, the corresponding page is not available in the main memory. Such condition is referred to as **page fault.** When page fault occurs, the executing of the

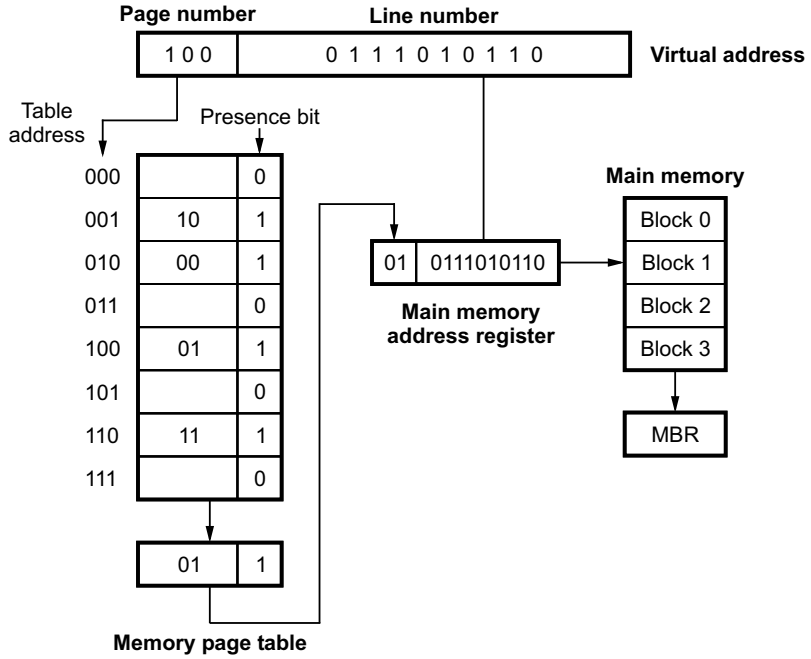


Fig. 4.9.2 Memory table in the paged system

present program is suspended until the required page is copied from auxiliary memory into the main memory. If main memory is already full, it is necessary to remove a page from a block of main memory to create a space for the new page. The policy for choosing pages to remove is determined by the replacement algorithm.

4.9.3 Page Replacement Algorithms

There are many different page replacement algorithms used by various operating systems. Let us discuss some of them.

FIFO Algorithm

It is the simplest page-replacement algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (6, 0, 1) cause page faults, and the required pages are brought into these empty frames.

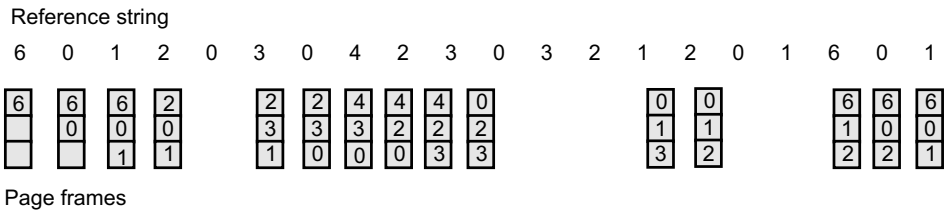


Fig. 4.9.3 FIFO page-replacement algorithm

The next reference (2) replaces page 6, because page 6 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1 and 2) to be brought in. This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Fig. 6.41. Every time a fault occurs, we show which pages are in out three frames. There are 15 faults algorithm.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. It may replace the most needed page as it is oldest page.

Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. It has been called OPT or MIN. It simply states that "*Replace the page that will not be used for the longest period of time*". Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page replacement algorithm would yield nine page faults, as shown in Fig. 4.9.4. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 6, because 6 will not be used until reference 18, whereas page 0 will be used at 5 and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

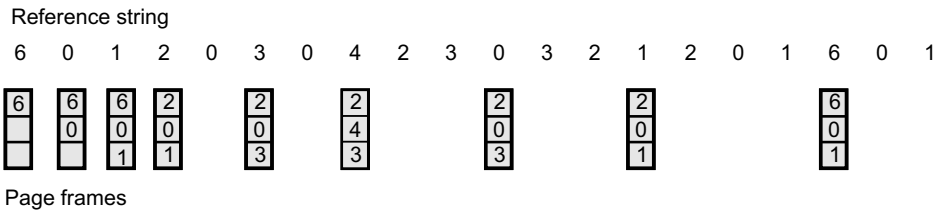


Fig. 4.9.4 Optimal page replacement algorithm

Unfortunately, the optimal page replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Algorithm

The algorithm which replaces the page that has not been used for the longest period of time is referred to as the *Least Recently Used* (LRU) algorithm.

The result of applying LRU replacement to our example reference string is shown in Fig. 4.9.5. The LRU algorithm produces 12 faults. It can be noticed that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

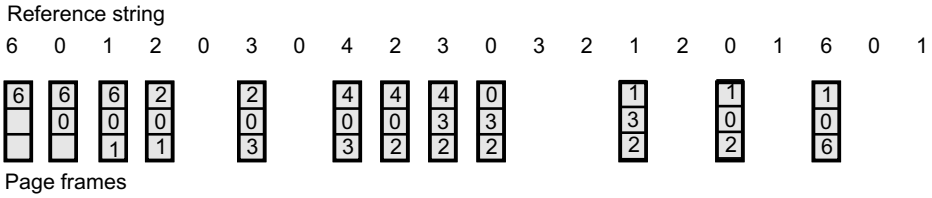


Fig. 4.9.5 LRU page replacement algorithm

Example 4.9.2 *A virtual memory system has an address space of 8 K words, a memory space of 4 K words and pages of 4 K words and page and block size of 1 K words. The following page references changes occur during a given time interval.*

4, 2, 0, 1, 2, 6, 1, 4, 0, 1, 0, 2, 3, 5, 7

Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm used is

i) LRU ii) LFU iii) FIFO

Solution : The Fig. 4.9.6 shows the address space and memory space split into groups of 1K words. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

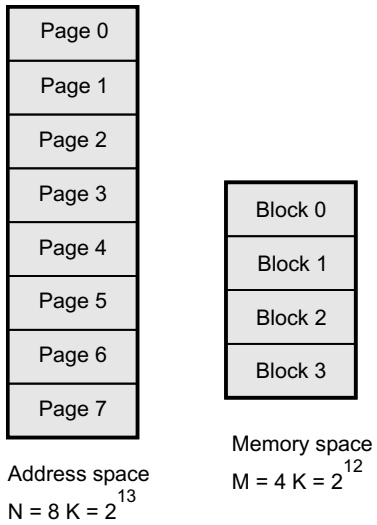
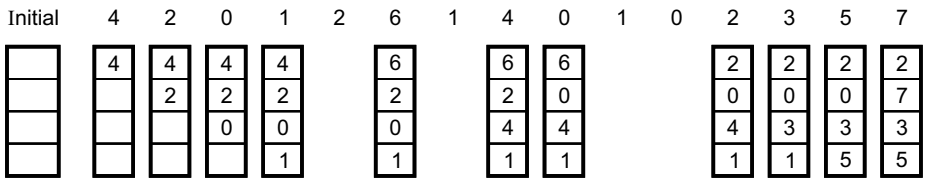
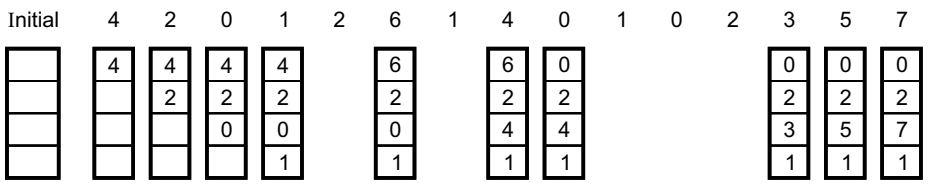


Fig. 4.9.6 Address space and memory space

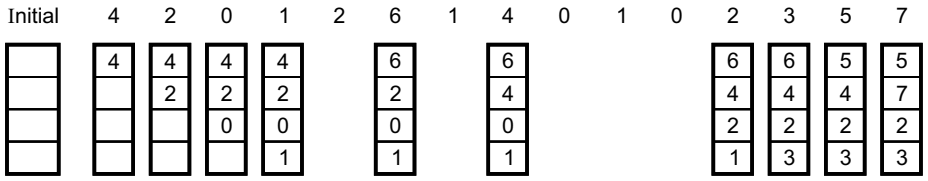
i) LRU : Least Recently Used



ii) LFU : Least Frequently Used



iii) FIFO : First in First Out



Example 4.9.3 *Consider a paging system in which M_1 has capacity of three pages. The execution of a program Q requires reference to nine distinct pages P_i, where i = 1, 2, ... 9, and I is the page address. The page address stream formed by executing Q is 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4 and 5. How many page faults would occur for the following replacement algorithms ?*

i) LRU ii) FIFO replacement iii) OPT. **May-11, Marks 8**

Solution : The Fig. 4.9.7 shows the manner in which the pages are assigned to M_1 using LRU, FIFO and OPT replacement policies.

1. LRU

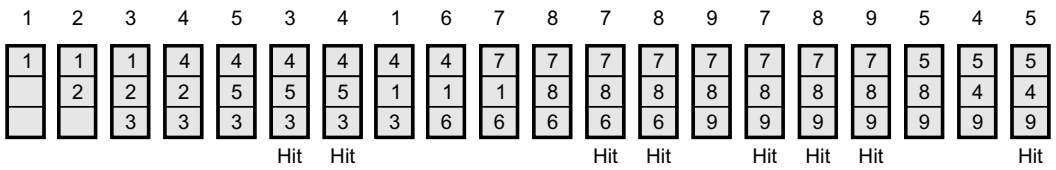


Fig. 4.9.7 (a)

2. FIFO

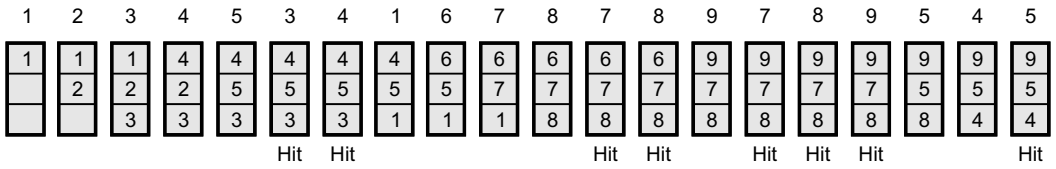


Fig. 4.9.7 (b)

3. OPT

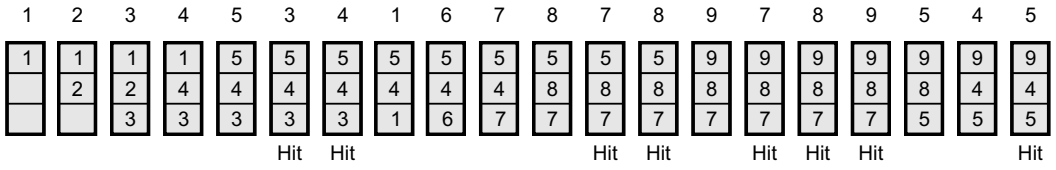


Fig. 4.9.7 (c)

Example 4.9.4 *A byte-addressable computer has a small data cache capable of holding eight 32-bit words. Each cache block contains one 32-bit word. When a given program is executed, the processor reads data from the following sequence of hex addresses -*

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4. The pattern is repeated 4 times. Assuming that the cache is initially empty, show the contents of the cache at the end of each pass and compute the hit rate for a direct mapped cache. **May-06, Marks 6**

Solution : Cache entries after first pass : **Hit** : 3 : **Miss** : 9

0	200	201	202	203
4	204	205	206	207
8	208	209	20A	20B
C	24C	24D	24E	24F
10	2F0	2F1	2F2	2F3
14	2F4	2F5	2F6	2F7
18	218	219	21A	21B
1C	21C	21D	21E	21F

The contents of cache after each pass will remain same. There will be 10 hits and 2 miss in 2nd, 3rd and 4th pass.

Therefore, in all there are 15 miss and 33 hits.

$$\text{Hit rate} = \frac{33}{48} \times 100 = 68.75\ \%$$

4.9.4 Memory Management Hardware

The Fig. 4.9.8 shows a typical memory organization that implements virtual memory. The **memory management unit** controls this virtual memory system. It translates virtual address into physical addresses. A simple method for translating virtual addresses into

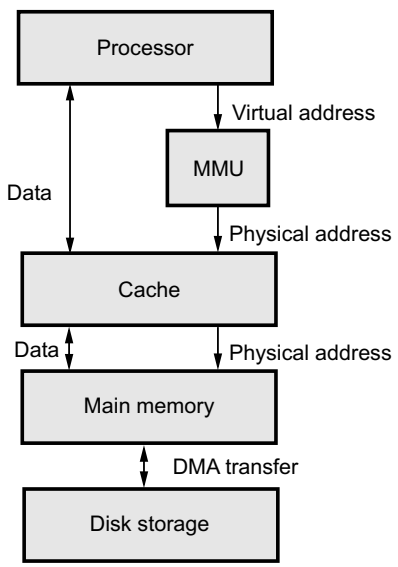


Fig. 4.9.8 Virtual memory organization

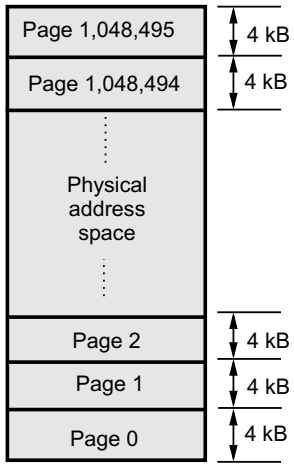


Fig. 4.9.9 Paged organization of the physical address space

physical addresses is to assume that all programs and data are composed of fixed length unit called *pages*, as shown in the Fig. 4.9.9. Pages constitute the basic unit of information that is moved between the main memory and the disk whenever the page translation mechanism determines that a swapping is required.

4.9.4.1 Segmented Page Mapping

The logical address specified by the program is partitioned into three fields : segment, page and word. The segment field specifies a segment number. The page field specifies the page within the segment and the word field gives the specific word within the page. This is illustrated in Fig. 4.9.10.

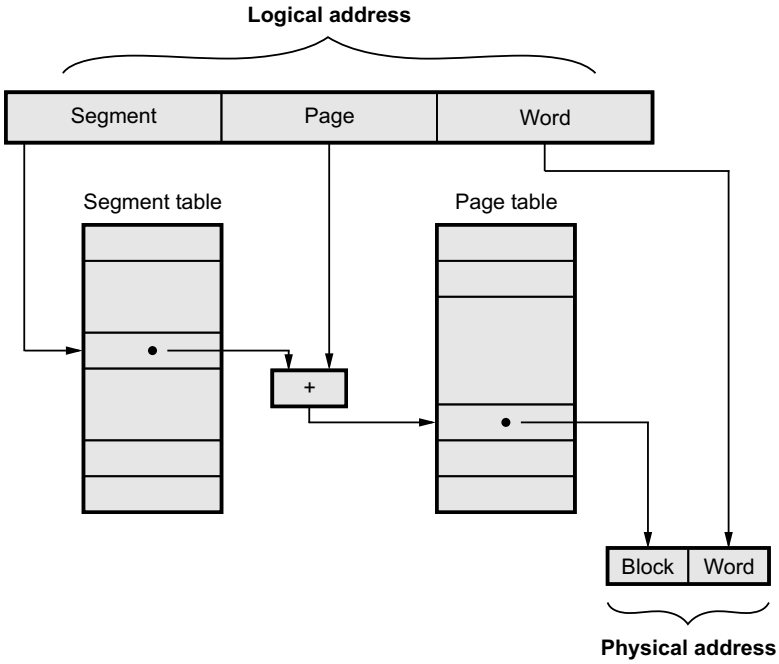


Fig. 4.9.10 Mapping of logical address into physical address

A page field of width K can specify up to 2^K pages. A segment number may be associated with just one page or with as many as 2^K pages. Thus the length of a segment varies according to the number of pages that are assigned to it.

As shown in the Fig. 4.9.10, two tables are used to map the logical address into a physical address. The segment number of the logical address specifies the address for the segment table. The entry in the segment table is a pointer address for a page table base. The page number given in the logical address is added to the page table base to give a pointer address to an entry in the page table. The value specified in the page

table provides the block number in the physical memory. The concatenation of the block field with the word field produces the final physical mapped address.

The two tables (segment table and page table) used in the mapping process are stored in either two separate memories or in main memory. In both the cases, a memory reference from the CPU requires three accesses to memory :

- One from the segment table,
- Second from the page table and
- Third from the main memory.

This process slows down the memory access. To avoid this speed penalty and to speed up memory access, the associative memory called **Translation Lookaside Buffer (TLB)** is used to hold the most recently referred table entries. When first time any block is referenced, its value together with the corresponding segment and page numbers are entered into the associative memory as shown in the Fig. 4.9.11. Thus in the mapping process the references are searched in the TLB.

If they are in TLB (Hit), the mapping delay is only that of the associative memory. If references are not in associative memory (Miss), the slower table mapping shown in Fig. 4.9.10 is used and the references are stored in the associative memory for future reference.

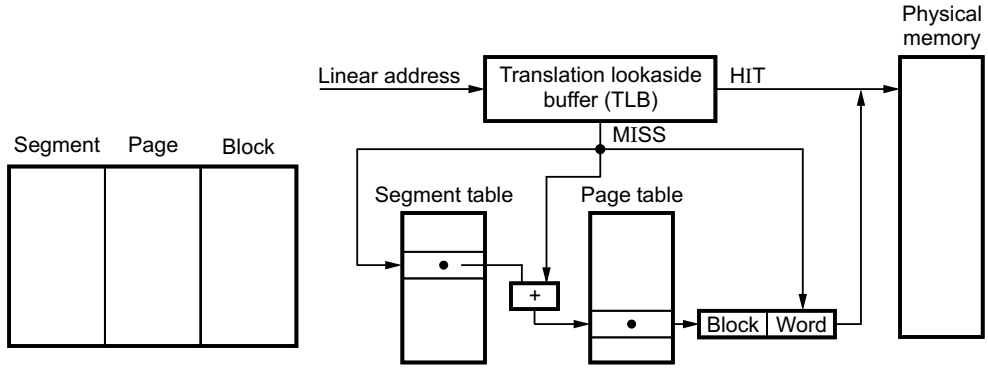


Fig. 4.9.11

Example 4.9.5 *The logical address space in a computer system consists of 128 segments. Each segment can have up to 32 pages of 4 K words each. Physical memory consists of 4 K blocks of 4 K words in each. Formulate the logical and physical address formats.*

Solution : Number of bits for segment address = $\log_2 128 = \log_2 2^7$ = **7 bits**

Number of bits for page address = $\log_2 32 = \log_2 2^5$ = **5 bits**

Number of bits for word address = $\log_2 4096 = \log_2 2^{12}$ = **12 bits**

Number of bits for block address = $\log_2 4096 = \log_2 2^{12}$ = **12 bits**

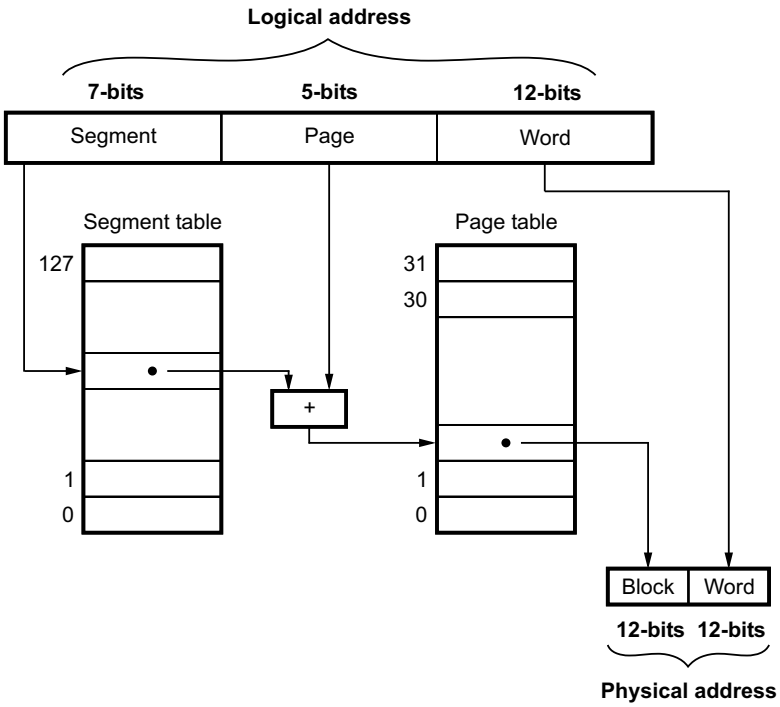


Fig. 4.9.12

4.9.4.2 Memory Protection

There is a need for memory protection for the following reasons :

- To prevent two or more users from reading and changing the contents of a memory location at the same time.
- To prevent user programs from corrupting the operating system code and data areas.

We can protect memory in two ways : By providing protection information to the physical address or the logical address. In case of physical address, a number of protection bits are assigned to each block in memory to indicate the type of access allowed to its corresponding block. Every time, when page is moved from one block to another it is necessary to update protection bits accordingly.

Much better approach is to apply protection in the logical address space rather than physical address space. In this case, the protection information is included within the segment table or segment register of the memory management hardware.

The content of each entry in the segment table or a segment register is called a segment **descriptor**. In addition to base address and the length, the segment descriptor contains the protection information. The Fig. 4.9.13 shows the typical format of a segment descriptor. The base address field in the descriptor gives the base of the page table

address in a segmented-page organization or the block base address in a segment register organization. This address is used to map from a logical address to the physical address. The length field gives the segment size by specifying the maximum number of pages assigned to the segment. The length field is compared against the page number in the logical address.

The protection field in the descriptor or entry in the page table specifies the access rights available to the particular segment. Some of the access rights used for protection are :

1. Full read and write privileges
2. Read only access (write protection)
3. Execute only (program protection)
4. System only (operating system protection)

Full read and write privileges are given to the program when it is executing its own instructions. Write protection is given to the system programs such as utility program and library programs.



Fig. 4.9.13 Format of a typical segment descriptor

The execute only protection prevents programs from being copied. The operating system protection prevents access of operating system programs to unauthorized users.

Review Questions

1. *What is the function of a TLB (translation look-aside buffer) ?* **May-06, Marks 2**
2. *Discuss the address translation mechanism and the different page replacement policies used in a virtual memory system.* **May-06, Marks 10**
3. *Discuss the address translation mechanism.* **May-06, Marks 5**
4. *Explain the virtual memory address translation and TLB with necessary diagram.* **Dec.-06, Marks 10**
5. *What is virtual memory ? Explain how the logical address is translated into physical address in the virtual memory system with a neat diagram.* **May-07, Marks 10**
6. *Explain the concept of virtual memory with any one virtual memory management technique.* **May-07, Marks 8**
7. *What is virtual memory ? How is it implemented ?* **Dec.-07, Marks 2**
8. *Explain how the virtual address is converted into real address in a paged virtual memory system.* **May-08, Marks 10**

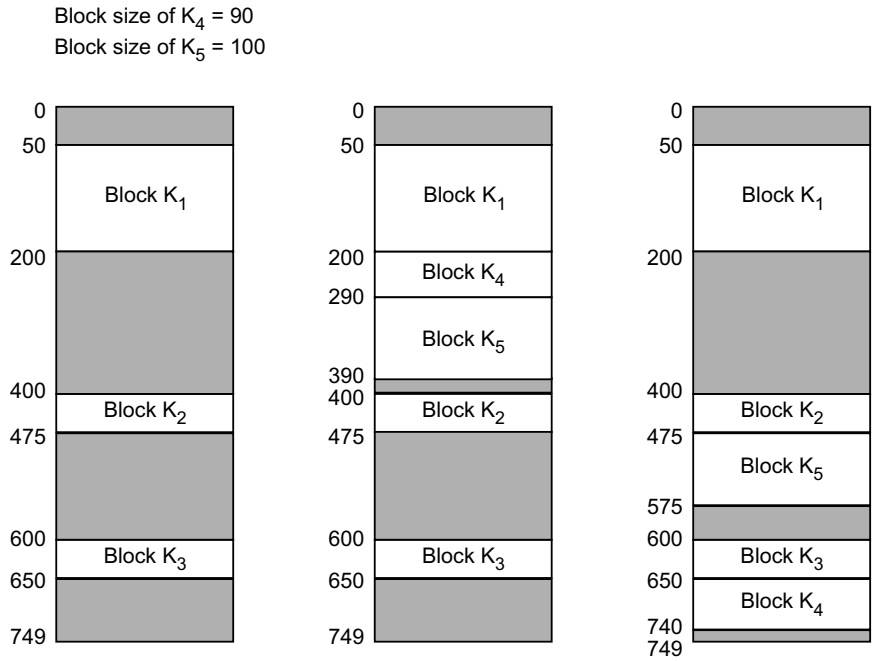
9. *How does a virtual address gets translated into a physical address. Explain in detail with a neat diagram. Explain the use of TLB.* **May-08, Marks 16**
10. *What is virtual memory ? Explain the various memory management schemes. What is the need for different memory management schemes ?* **Dec.-08, Marks 16**
11. *Explain about virtual memory and its management.* **Dec.-08, Marks 16**
12. *Explain the virtual memory address translation and TLB with necessary diagram.* **Dec.-08, 10, May-09, 10, Marks 10**
13. *Explain how the virtual address is translated into physical address in reduce paged memory system with the help of a diagram.* **May-09, Marks 6**
14. *What is a virtual memory ?* **May-09, Dec.-12, Marks 8**
15. *What is TLB ?* **May-10, Marks 2**
16. *What are the advantages of using virtual memory ?* **Dec.-10, Marks 2**
17. *Explain how logical address is translated into real address in segmentation and paging system.* **May-11, Marks 8**

4.10 Memory Allocation

May-11

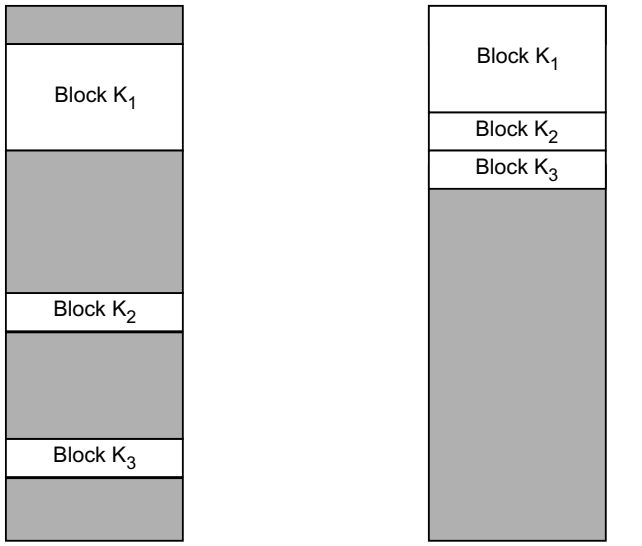
In a memory system, the various levels of a memory are divided into sets of contiguous locations which are called regions or segments or pages. These contiguous locations store blocks of data. Blocks are swapped automatically among the levels in order to minimise the access time seen by the processor. Swapping generally initiated in response to process request. The placement of blocks of information in a memory system is called **memory allocation**. To allocate memory it is necessary to keep information of available memory in the system. The memory management system keeps this information in the table. This table indicates which parts of memory are available and which are occupied. When a new program needs memory, memory management system searches for a available memory such that it is large enough to accommodate this program. If memory management system finds sufficient free memory, it allocates only as much memory as is needed, keeping the rest available to satisfy future requests. If sufficient free memory is not available, swapping of blocks is done. The blocks which are no longer required are deallocated and the regions occupied by them are transferred from the occupied space list to the available space list. The criteria for selecting a particular block to be replaced is decided by the **replacement policy**. Unused space between occupied regions is called **external fragmentation.** Unused space within a partially filled page frame is called **internal fragmentation.**

It is important to select the best suitable memory block for the allocation of particular program, because efficient memory allocation algorithm result in a high hit ratio and a low average access time. If the hit ratio is low, an excessive amount of swapping between memory levels occurs, a phenomenon known as **thrashing**. Efficient memory



(a) Initial memory state **(b) Allocation of K_4 and K_5 by first-fit** **(c) Allocation of K_4 and K_5 by best-fit**

Fig. 4.10.1 Nonpreemptive memory allocation



(a) Memory allocation before compaction **(b) Memory allocation after compaction**

Fig. 4.10.2

allocation also minimizes the amount of unused or underused space in the memory. Two methods are used for dynamic memory allocation : Nonpreemptive allocation and preemptive allocation.

Nonpreemptive allocation : Consider M_1 as a main memory and M_2 as a secondary memory and a block k_i of n_i words is to be transferred from M_2 to M_1. For such memory allocation it is necessary to find or create an available region of n_i or more words to accommodate k_i. This process is known as **nonpreemptive allocation**. Two algorithms commonly used algorithms in nonpreemptive allocation are : First-fit and Best-fit.

First fit : In this algorithm, searching is started either at the beginning of the memory or where the previous first-fit search ended. In this algorithm the first free memory block which is big enough is allocated to the block k_i. The searching process is stopped as soon as a free memory block with enough space is allocated.

Best fit : In this algorithm, all free memory blocks are searched and smallest free memory block which is large enough to accommodate desired k_i Block is used to allocate k_i. This algorithm uses free memory space more efficiently than first-fit algorithm. The Fig. 4.10.1 shows the allocation of memory blocks using first fit and best fit algorithms.

Preemptive allocation : Nonpreemptive allocation cannot make efficient use of memory in all situations. Due to scattered memory blocks larger free memory blocks may not be available. Much more efficient use of the available memory space is possible if the occupied space can be reallocated to make a room for incoming blocks. Reallocation of the blocks can be done by a method called **compaction.** This is shown in the Fig. 4.10.2. In compaction, the blocks currently in the memory are compressed into a single contiguous group at one end of the memory. This creates an available region of maximum size. Once the memory is compacted, incoming blocks are assigned to contiguous regions at the unoccupied end. In this algorithm, the single large memory block is available for allocation thus block selection procedure is not required. However, this algorithm requires long compaction time.

Comparison between paging and segmentation

Sr. No.	Segmentation	Paging
1.	Program is divided into variable size segments.	Program is divided into fixed size pages.
2.	User (or compiler) is responsible for dividing the program into segments.	Division into pages is performed by the operating system.

3.	Segmentation is slower than paging.	Paging is faster than segmentation.
4.	Segmentation is visible to the user.	Paging is invisible to the user.
5.	Segmentation eliminates internal fragmentation.	Paging suffers from internal fragmentation.
6.	Segmentation suffers from external fragmentation.	There is no external fragmentation.
7.	Processor uses page number, offset to calculate absolute address.	Processor uses segment number, offset to calculate absolute address.
8.	OS maintain a list of free holes in main memory.	OS must maintain a free frame list.

Review Questions

1. *Write a short note on memory allocation.*
2. *Compare nonpreemptive and preemptive memory allocation.*
3. *Describe preemptive and non-preemptive memory allocation in detail.* **May-11, Marks 8**

4.11 Associative Memory

May-07, 11, Dec.-03, 12

Many data-processing applications require the search of items in a table stored in memory. They use object names or number to identify the location of the named or numbered object within a memory space. For example, an account number may be searched in a file to determine the holder's name and account status. To search an object, the number of accesses to memory depends on the location of the object and the efficiency of the search algorithm.

The time required to find an object stored in memory can be reduced considerably if objects are selected based on their contents, not on their locations. A memory unit accessed by the content is called an **associative memory** or **Content Addressable Memory (CAM)**. This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location.

4.11.1 Hardware Organization

The Fig. 4.11.1 shows the block diagram of an associative memory. It consists of memory array with match logic for m n-bit words and associated registers. The argument register (A) and key register (K) each have n-bits per word. Each word in memory is compared in parallel with the contents of the argument register. The words that match with the word stored in the argument register set a corresponding bits in the

match register. Therefore, reading can be accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or bits in the argument word. Only those bits in the argument register having 1's in their corresponding position of the key register are compared. For example, if argument register A and the key register K have the bit configuration shown below. Only the three rightmost bits of A are compared with memory words because K has 1's in these positions.

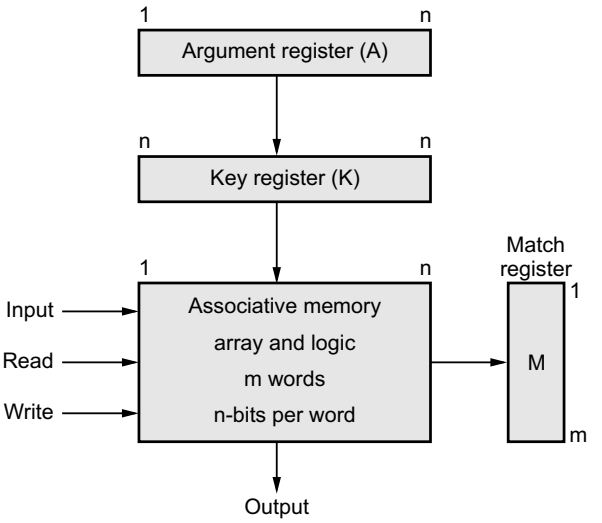


Fig. 4.11.1 Block diagram of associative memory

A	1 1 0 1 1	**0 1 0**
K	0 0 0 0 0	**1 1 1**
Word 1	0 1 0 1 0	**0 1 0** match
Word 2	1 1 0 1 1	**1 0 0** no match

The Fig. 4.11.2 shows the associated memory with cells of each register. The cells in the memory array are marked by the letter C with two subscripts. The first subscript gives the word number and the second subscript gives the bit position in the word.

The Fig. 4.11.3 shows the internal organization of a typical cell. It consists of RS flip-flop as a storage element and the circuit for reading, writing and matching the cell. By making the write signal logic 1, it is possible to transfer the input bit into the storage cell. To carry-out read operation read signal is made logic 1. The match logic compares the bit in the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the corresponding bit in the match register.

The Fig. 4.11.4 shows the match logic for each word. The cell bit (Q_{ij}) is compared with the corresponding argument bit with EX-NOR gate. EX-NOR gate gives output logic 1 only when its inputs are same. The output of EX-NOR gate is valid only when the corresponding bit in key register is logic 1. This condition is implemented using 2-input OR gate. When corresponding bit in key register is logic 1, the inverter gives output 0 which forces the output of OR gate to follow the second input, i.e. the

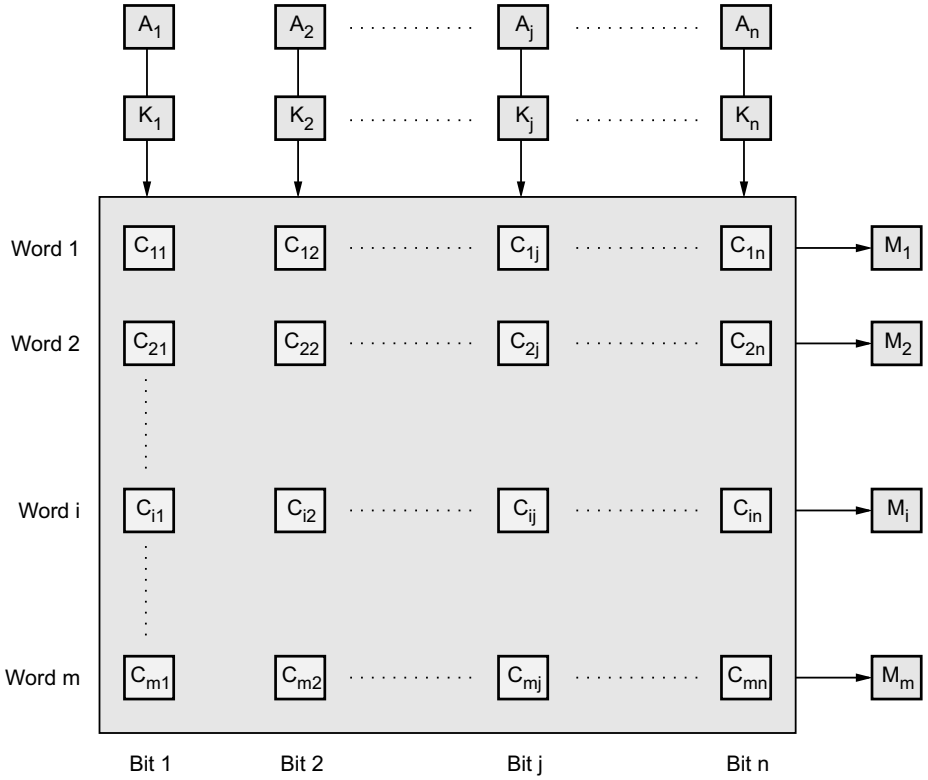


Fig. 4.11.2 Associated memory showing cells of each register

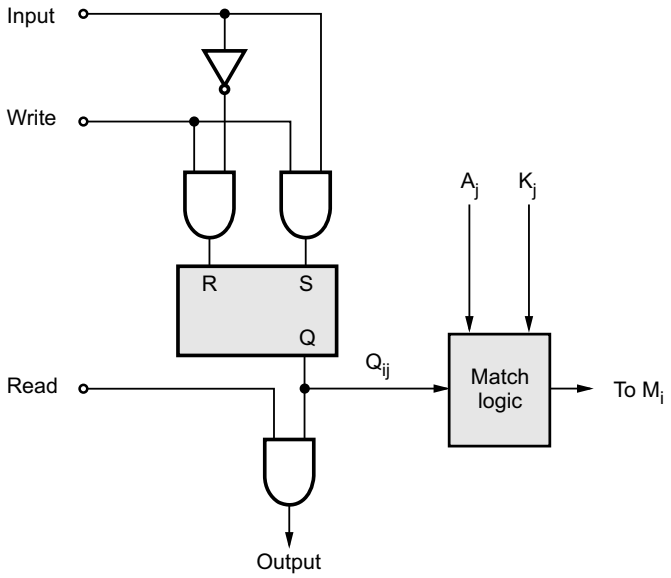


Fig. 4.11.3 Internal organization of typical cell of associative memory

comparison output; otherwise output of OR-gate is logic 1. The outputs of all OR-gates are then ANDed with n-input AND gate to check whether all bits in the word are matched with the bits in the argument register.

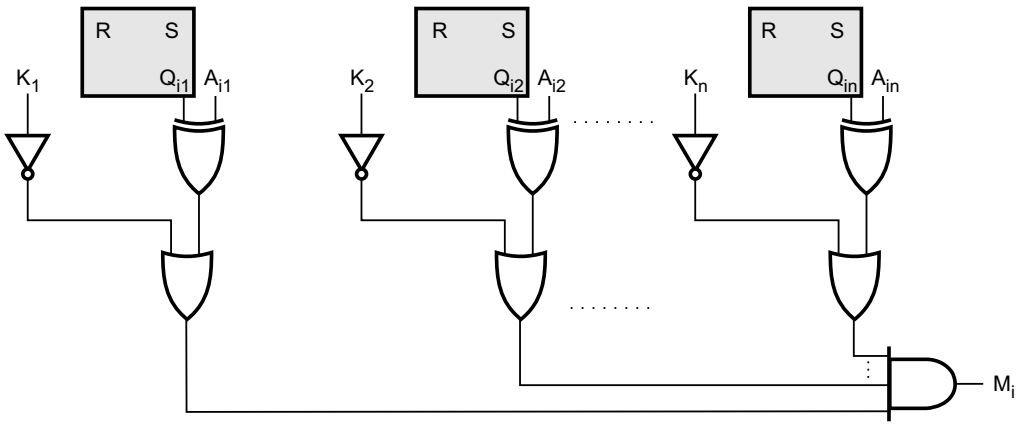


Fig. 4.11.4 Match logic for one word of associative memory

4.11.2 Read Operation

In the read operation, all matched words are read in sequence by applying a read signal to each word line whose corresponding M_i bit is a logic 1. In applications where no two identical items are stored in the memory, only one word may match the unmasked argument field. In such case, we can use M_i output directly as a read signal for the corresponding word. The contents of the matched word will be presented automatically at the output lines and no special read signal is needed.

4.11.3 Write Operation

In write operation, we consider two separate cases : 1. It is necessary to load entire memory with new information 2. It is necessary to replace one word with new information. The entire memory can be loaded with new information by addressing each location in sequence. This will make the memory device a random access memory for writing and a content addressable memory for reading. Here, the address for the input can be decoded as in a random access memory. Thus instead of having m address lines, one for each word, the number of address lines can be reduced by the use of decoder to d lines, where $m = 2^d$.

To implement the write operation in the second case the tag register is used. This register has as many bits as there are words in the memory for every active (valid) word stored in the memory, the corresponding bit in the tag register is set to 1. When word is deleted from the memory the corresponding tag bit is cleared, i.e. it is set to logic 0. The word is then stored in the memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a

new word. After the new word is stored in the memory it is made active by setting the corresponding tag bit in the tag register.

Review Questions

1. *Give the internal organisation of a typical cell of associative memory.* **Dec.-03, Marks 2**
2. *Describe in detail about associative memory.* **Dec.-03, Marks 16**
3. *Give the basic cell of an associative memory and explain its operation. Show how associative memories can be constructed using this basic cell with match logic* **May-07, Dec.-12, Marks 8**
4. *What is content addressable memory ? Explain with a help of a block diagram how multiple matched words can be read out from content addressable memory.* **May-11, Marks 8**

4.12 Memory Interleaving

Dec.-06, 07

We know that, the memory access time is the bottleneck in the system and one way to reduce it is to use cache memory. Alternative technique to reduce memory access time is **memory interleaving**. In this technique, the main memory is divided into a number of memory modules and the addresses are arranged such that the successive words in the address space are placed in different modules. Refer Fig. 4.12.1. Most of the times CPU accesses consecutive memory locations. In such situations address will be to the different modules. Since these modules can be accessed in parallel, the average access time of fetching word from the main memory can be reduced.

The low-order k bits of the memory address are generally used to select a module, and the high-order m bits are used to access a particular location within the selected module. In this way, consecutive addresses are located in successive modules. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at any one time. This results in both faster access to a block of data and higher average utilization of the memory system as a whole. It is important to note that to implement the interleaved memory structure, there must be 2^k modules; otherwise, there will be gaps of nonexistent locations in the memory address space.

The effect of interleaving is substantial. However, it does not speed up memory operation by a factor equal to the number of the module. It reduces the effective memory cycle time by a factor close to the number of modules.

Pipeline and vector processors often require simultaneous access to memory from two or more sources. This need can be satisfied by interleaved memory. A CPU with instruction pipeline can also take the advantage of interleaved memory modules so that each segment in the pipeline can access memory independent of memory access from other segments.

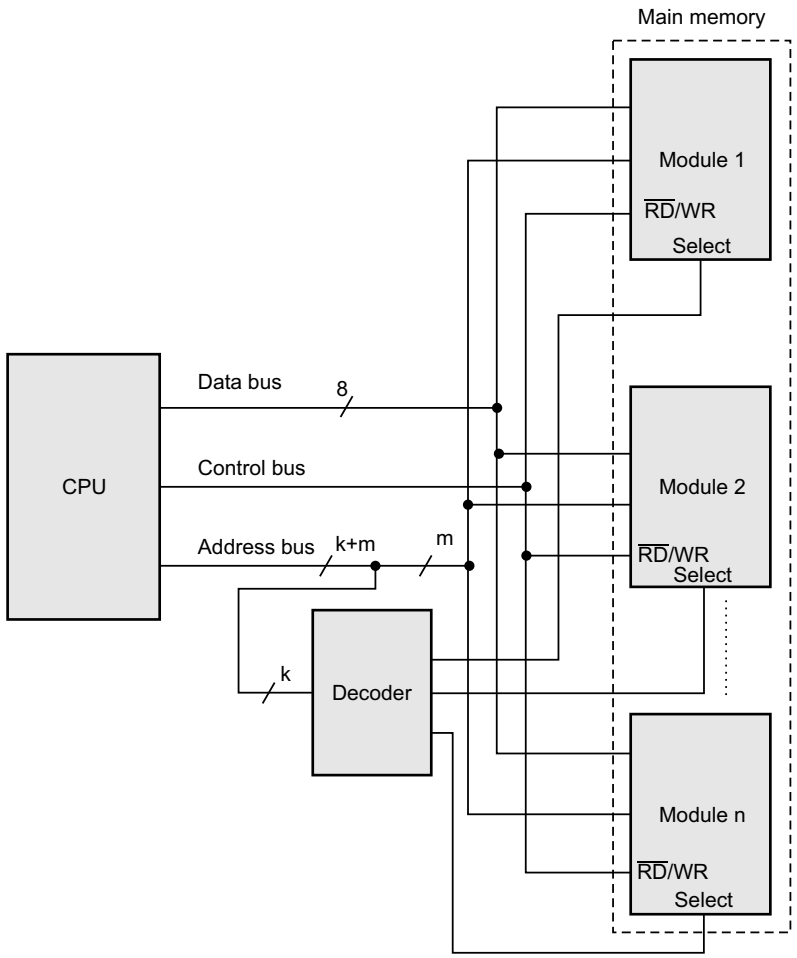


Fig. 4.12.1 Interleaved memory organization

Review Questions

1. *Discuss the concept of memory interleaving and give its advantages.* **Dec.-06, Marks 10**
2. *Write note on memory interleaving.* **Dec.-07, Marks 4**

Two Marks Questions with Answers

Q.1 ***Name the two types of storage devices.***

Ans. : The two types of storage devices are :

1. Primary memory
2. Secondary memory

Q.2 *Name the two types of RAM.*

Ans. : The two types of RAM are :

1. Static RAM 2. Dynamic RAM

Q.3 *Distinguish between static RAM and dynamic RAM ?* **May-08, 11**

Ans. : Refer section 4.3.3.

Q.4 *Distinguish between asynchronous DRAM and synchronous RAM.* **Dec.-08**

Ans. : Refer section 4.5.3.

Q.5 *What are static memories ?* **May-12**

Ans. : Memories that consists of circuits capable of retaining their state as long as power is applied are known as **static memories**.

Q.6 *What are static and dynamic memories ?*

Ans. : Static memory are memories which require periodic no refreshing. Dynamic memories are memories, which require periodic refreshing.

Q.7 *Define memory latency.*

Ans. : The term memory latency is used to refer to the amount of time it takes to transfer a word of data to or from the memory. The term latency is used to denote the time it takes to transfer the first word of data. This time is usually substantially longer than the time needed to transfer each subsequent word of a block.

Q.8 *Define memory bandwidth.*

Ans. : Memory bandwidth is a product of the rate at which the data are transferred (and accessed) and the width of the data bus.

Q.9 *What will be the width of address and data buses for a 512 K × 8 memory chip ?* **Dec.-07**

Ans. : The widths of address and data buses are 19 and 8, respectively.

Q.10 *Give the features of a ROM cell.* **May-08**

Ans. : The main features of a ROM cell are :

- It can hold one bit data.
- It can hold data even if power is turned off.
- We can't write data in ROM cell; it is read only.

Q.11 *Name the different types of ROMs.*

Ans. : There are four types of ROMs are :

- Masked ROM

- PROM
- EPROM and
- EEPROM or E^2PROM.

Q.12 *What is the function of memory controller ?*

Ans. : The memory controller is connected between processor and the DRAM memory. It does the task of generating multiplexed address as well as task of generating control signals such as $\overline{RAS}$, $\overline{CAS}$, R/$\overline{W}$, $\overline{CS}$ and so on for DRAM.

Q.13 *What is refreshing overhead ?*

Ans. : $$\text{Refreshing overhead} = \frac{\text{Time required for refresh}}{\text{Interval time}}$$

Q.14 *Compare SDRAM with DDR SDRAM.* **May-09**

Ans. : The standard SDRAM performs its operations on the rising edge of the clock signal. On the other hand, the DDR SDRAMs transfers data on both the edges of the clock signal. The latency of the DDR SDRAMs is same as that for standard SDRAM. However, since they transfer data on both the edges of the clock signal, their bandwidth is effectively doubled for long burst transfer.

Q.15 *What is cache memory ?*

Ans. : In the memory system small section of SRAM is added along with main memory, referred to as **cache memory**.

Q.16 *Define hit rate.*

Ans. : The percentage of accesses where the processor finds the code or data word it needs in the cache memory is called the hit rate.

Q.17 *Define miss rate and miss penalty.*

Ans. : The percentage of accesses where the processor does not find the code or data word it needs in the cache memory is called the miss rate. Extra time needed to bring the desired information into the cache is called the miss penalty.

Q.18 *What is program locality ?*

Ans. : In cache memory system, prediction of memory location for the next access is essential. This is possible because computer systems usually access memory from the consecutive locations. This prediction of next memory address from the current memory address is known as **program locality**. Program locality enables cache controller to get a block of memory instead of getting just a single address.

Q.19 *Define locality of reference. What are its types ?* **May-08**

Ans. : The program may contain a simple loop, nested loops, or a few procedures that repeatedly call each other. The point is that many instructions in localized area of the program are executed repeatedly during some time period and the remainder of the program is accessed relatively infrequently. This is referred to as **locality of reference**.

Q.20 *Define the terms : Spatial locality and temporal locality.* **May-08, Dec.-12**

Ans. : Locality of reference manifests itself in two ways : Temporal and spatial. The temporal means that a recently executed instruction is likely to be executed again very soon. The spatial means that instructions stored near by to the recently executed instruction are also likely to be executed soon.

Q.21 *Explain the concept of block fetch.*

Ans. : Block fetch technique is used to increase the hit rate of cache. A block fetch can retrieve the data located before the requested byte (look behind) or data located after the requested byte (look ahead), or both. When CPU needs to access any byte from the block, entire block that contains the needed byte is copied from main memory into cache.

Q.22 *Name the common replacement algorithms.*

Ans. : The four most common replacement algorithms are :

- Least-Recently Used (LRU)
- First-In-First-Out (FIFO)
- Least-Frequently-Used (LFU)
- Random

Q.23 *What is a mapping function ?* **May-08**

Ans. : Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a **mapping function.**

Q.24 *List the mapping techniques.*

Ans. : There are two main mapping techniques which decides the cache organisation :

1. Direct-mapping technique
2. Associative-mapping technique

The associative mapping technique is further classified as fully associative and set associative techniques.

Q.25 *What do you mean associative mapping technique ?*

Ans. : The tag of an address received from the CPU is compared to the tag bits of each block of the cache to see if the desired block is present. This is called associative mapping technique.

Q.26 *What is the need of cache updating ?*

Ans. : In a cache system, two copies of same data can exist at a time, one in cache and one in main memory. If one copy is altered and other is not, two different sets of data become associated with the same address. To prevent this cache updating is needed.

Q.27 *Name the different cache updating systems.*

Ans. : The different cache updating systems are :

- Write through system
- Buffered write through system and
- Write-back system.

Q.28 *What is memory interleaving ?*

Ans. : The memory interleaving is a technique to reduce memory access time by dividing memory into a number of memory modules and the addresses are arranged such that the successive words in the address space are placed in different modules. Most of the times CPU accesses consecutive memory locations. In such situations accesses will be to the different modules. Since these modules can be accessed in parallel, the average access time of fetching word from the main memory can be reduced.

Q.29 *What is virtual memory ?* **Dec.-07, 08, May-07, 09, 10**

Ans. : In modern computers, the operating system moves program and data automatically between the main memory and secondary storage. Techniques that automatically swaps program and data blocks between main memory and secondary storage device are called **virtual memory**. The addresses that processor issues to access either instruction or data are called **virtual** or **logical address**.

Q.30 *What is the function of memory management unit ?*

Ans. : The memory management unit controls this virtual memory system. It translates virtual address into physical address.

Q.31 *What is TLB (Translation Look-aside Buffer) ?* **May-10**

Ans. : To support demand paging and virtual memory processor has to access page table which is kept in the main memory. To avoid the access time and degradation of performance, a small portion of the page table is accommodated in the memory management unit. This portion is called **Translation Lookaside Buffer (TLB)**.

Q.32 *What is the function of a TLB (Translation Look-aside Buffer) ?* **May-06**

Ans. : TLB is used to hold the page table entries that corresponds to the most recently accessed pages. When processor finds the page table entries in the TLB it does not have to access page table and saves substantial access time.

Q.33 *What is associative memory or Content Addressable Memory (CAM) ?*

Ans. : The time required to find an object stored in memory can be reduced considerably if objects are selected based on their contents, not on their locations. A memory unit accessed by the content is called an **associative memory** or **Content Addressable Memory (CAM)**.

Q.34 *What is the key feature of Rambus ?*

Ans. : The key feature of Rambus technology is a fast signaling method used to transfer information between chips.

Q.35 *What is Rambus channel ?*

Ans. : The differential signaling and high transmission rates can be achieved with the use of special interface circuits and communication links. Rambus provides a complete specification for the design of such communication link, called the **Rambus channel**.

Q.36 *What is the seek time ?*

Ans. : The **seek time** is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. The average seek time is in the range of 5 - 8 ms.

Q.37 *What is SCSI bus and SCSI drive ?*

Ans. : A disk drive is usually connected to the computer using standard interconnection scheme called **SCSI bus**. A disk drive along with the required SCSI interface circuitry is known as **SCSI drive**.

Q.38 *What is optical memory ?*

Ans. : Optical or light based techniques for data storage, such memories usually employ optical disk which resemble magnetic disk in that they store binary information in concentric tracks on an electromechanically rotated disks. The information is read as or written optically, however with a laser replacing the read write arm of a magnetic disk drive. Optical memory offer high storage capacities but their access rate is are generally less than those of magnetic disk.

Q.39 *List some applications for WORM.*

Ans. : Some of the application or WORM devices are :

- On-Line catalogs such as automobile party's dealer
- Large volume distribution
- Transaction logging such as stock trading company
- Multimedia archival.

Q.40 *What do you mean by disk spanning ?*

Ans. : Disk spanning is a method of attaching drives to a single host adapter. All drives appear as a single contiguous logical unit. Data is written to the first drive first and when the drive is full, the controller switches to the second drive, then the second drive writes until its full.

Q.41 *List some objectives for using RAID Systems.*

Ans. : RAID systems are used to meet the following objectives

- Hot backup of disk systems
- Large volume storage at lower cost
- Higher performance at lower cost
- Ease of data recovery
- High MTBF.

Q.42 *What are the different levels RAID ?*

Ans. : There are six discrete levels of RAID functionality. They are

- Level 0 - Disk striping
- Level 1 - Disk mirroring
- Level 2 - Bit interleaving of data
- Level 3 - Bit interleaving with dedicated parity drives
- Level 4 - Sector interleaving of data with dedicated parity drive
- Level 5 - Block interleaving of data.

Q.43 *What is data stripping ?* **Dec.-09**

Ans. : In a RAID system, a single large file is stored in several separate disk units by breaking the file up into a number of smaller pieces and storing these pieces on different disks. This is called **data stripping.**

Q.44 *How do you construct a 8 M × 32 memory using 512 K × 8 memory chips ?* **May-07**

Ans. : By connecting four 512 × 8 chips in parallel we can expand the word size to 32 and such sixteen blocks consisting of four memory chips we can construct 8 M × 32 memory.

Q.45 *Write the two advantages of MOS devices.* **Dec.-07**

Ans. : MOS devices have following advantages :

1. High density 2. High speed 3. Low power consumption.

Q.46 *Calculate the effective address time if average page-fault service time of 20 ms and a memory access time of 80 ns. Let us assume the probability of a page fault 10 %.* **Dec.-09**

Ans. : Refer example 4.9.5.

Q.47 *What are the characteristics of semiconductor RAM memories ?*

Ans. : They are available in a wide range of speeds.

Their cycle time range from 100 ns to less than 10 ns.

They replaced the expensive magnetic core memories.

They are used for implementing memories.

Q.48 *Why SRAMs are said to be volatile ?*

Ans. : SRAMs are said to be volatile because their contents are lost when power is interrupted.

Q.49 *What are the characteristics of SRAMs ?*

Ans. : i) SRAMs are fast.

ii) They are volatile.

iii) They are of high cost.

iv) Less density.

Q.50 *What are the characteristics of DRAMs ?*

Ans. : i) Low cost.

ii) High density.

iii) Refresh circuitry is needed.

Q.51 *Define refresh circuit.*

Ans. : Refresh circuit is a circuit which ensures that the contents of a DRAM are maintained when each row of cells are accessed periodically.

Q.52 *What are asynchronous DRAMs ?* **Dec.-12**

Ans. : In asynchronous DRAMs, the timing of the memory device is controlled asynchronously. A specialized memory controller circuit provides the necessary control signals RAS and CAS that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are asynchronous DRAMs.

Q.53 *What are synchronous DRAMs ?*

Ans. : Synchronous DRAMs are those whose operation is directly synchronized with a clock signal.

Q.54 *Define memory access time.*

Ans. : The time required to access one word is called the memory access time. Or It is the time that elapses between the initiation of an operation and the completion of that operation.

Q.55 *Define memory cycle time.*

Ans. : It is the minimum time delay required between the initiation of two successive memory operations. That is the time between two successive read operations.

Q.56 *Define memory cell.*

Ans. : A memory cell is capable of storing one bit of information. It is usually organized in the form of an array.

Q.57 *What is a word line ?*

Ans. : In a memory cell, all the cells of a row are connected to a common line called as word line.

Q.58 *What is double data rate SDRAMs ?*

Ans. : Double data rates SDRAMs are those which can transfer data on both edges of the clock and their bandwidth is essentially doubled for long burst transfers.

Q.59 *What is motherboard ?*

Ans. : Motherboard is a main system printed circuit board which contains the processor.

Q.60 *What are SIMMs and DIMMs ?*

Ans. : SIMMs are Single In-line Memory Modules. DIMMs are Dual In-line Memory Modules. Such modules are an assembly of several memory chips on a separate small board that plugs vertically into a single socket on the motherboard.

Q.61 *What is Rambus technology ?*

Ans. : The key feature of Rambus technology is a fast signaling method used to transfer information between chips. Instead of using signals that have two voltage levels to represent the logic values, the signals consist of much smaller voltage swings around a reference voltage, vref. Small voltage swings make it possible to have short transition times, which allows for a high speed of transmission.

Q.62 *What are RDRAMs ?*

Ans. : RDRAMs are Rambus DRAMs. Rambus requires specially designed memory chips. These chips use cell arrays based on the standard DRAM technology. Multiple banks of cell arrays are used to access more than one word at a time. Circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as RDRAMs.

Q.63 *What are the special features of Direct RDRAMs ?*

Ans. : i) It is a two channel Rambus.

ii) It has 18 data lines intended to transfer two bytes of data at a time.

iii) There are no separate address lines.

Q.64 *What are RIMMs ?*

Ans. : RDRAM chips can be assembled in to larger modules called RIMMs. It can hold upto 16 RDRAMs.

Q.65 *Define ROM.*

Ans. : It is a non-volatile and read only memory.

Q.66 *What are the features of PROM ?*

Ans. : They are programmed directly by the user.

i) Faster

ii) Less expensive

iii) More flexible

Q.67 *Why EPROM chips are mounted in packages that have transparent window ?*

Ans. : Since the erasure requires dissipating the charges trapped in the transistors of memory cells. This can be done by exposing the chip to UV light .

Q.68 *What are the disadvantages of EPROM ?*

Ans. : The chip must be physically removed from the circuit for reprogramming and its entire contents are erased by the ultraviolet light.

Q.69 *What are the advantages and disadvantages of using EEPROM ?*

Ans. : The advantages are that EEPROMs do not have to be removed for erasure. Also it is possible to erase the cell contents selectively. The only disadvantage is that different voltages are needed for erasing, writing and reading the stored data.

Q.70 *Define flash memory.*

Ans. : Flash memories are read/write memories. In flash memories it is possible to read the contents of a single cell, but it is only possible to write an entire block of cells. A flash cell is based on a single transistor controlled by trapped charge.

Q.71 *Differentiate flash devices and EEPROM devices.*

Ans. :

Flash devices	EEPROM devices
It is possible to read the contents of a single cell, but it is only possible to write an entire block of cells.	It is possible to read and write the contents of a single cell.
Greater density which leads to higher capacity.	Relatively lower density.

Lower cost per bit.	Relatively more cost.
Consumes less power in their operation and makes it more attractive for use in portable equipments that is battery driven.	Consumes more power.

Q.72 *What is cache memory ?*

Ans. : In the memory system small section of SRAM is added along with main memory, referred to as **cache memory**.

Q.73 *What is a hit ?*

Ans. : A successful access to data in cache memory is called hit.

Q.74 *Define cache line.*

Ans. : Cache block is used to refer to a set of contiguous address locations of some size. Cache block is also referred to as cache line.

Q.75 *What are the two ways in which the system using cache can proceed for a write operation ?*

Ans. : i) Write through protocol technique.

ii) Write-back or copy back protocol technique.

Q.76 *What is write through protocol ?*

Ans. : In write through updating system, the cache controller copies data to the main memory immediately after it is written to the cache. Due to this main memory always contains a valid data, and any block in the cache can be overwritten immediately without data loss.

Q.77 *What is write-back or copy back protocol ?*

Ans. : In a write-back System, the alter bit in the tag field is used to keep information of the new data. Cache controller checks this bit before overwriting any block in the cache. If it is set, the controller copies the block to main memory before loading new data into the cache.

Q.78 *When does a readmiss occur ?*

Ans. : When the addressed word in a read operation is not in the cache, a read miss occur.

Q.79 *What is writemiss ?*

Ans. : During the write operation if the addressed word is not in cache then said to be writemiss.

Q.80 *What is load-through or early restart ?*

Ans. : When a read miss occurs for a system with cache the required word may be sent to the processor as soon as it is read from the main memory instead of loading into the cache. This approach is called load through or early restart and it reduces the processor's waiting period.

Q.81 *What is replacement algorithm ?*

Ans. : When the cache is full and a memory word that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the reference word. The collection of rules for making this decision constitutes the replacement algorithm.

Q.82 *What is virtual address ?*

Ans. : The binary address that the processor used for either instruction or data called virtual address.

Q.83 *What is virtual page number ?*

Ans. : Each virtual address generated by the processor whether it is for an instruction fetch is interpreted as a virtual page.

Q.84 *What is page frame ?*

Ans. : An area in the main memory that can hold one page is called page frame.

Q.85 *What is MMU ?*

Ans. : MMU is the Memory Management Unit. It is a special memory control circuit used for implementing the mapping of the virtual address space onto the physical memory.

Q.86 *What are pages ?*

Ans. : All programs and data are composed of fixed length units called pages. Each page consists of blocks of words that occupies contiguous locations in main memory.

Q.87 *What is dirty or modified bit ?*

Ans. : The cache location is updated with an associated flag bit called dirty bit.

Q.88 *Define access time for magnetic disks.*

Ans. : The sum of seek time and rotational delay is called access time for disks. Seek time is the time required to move the read/write head to the proper track. Rotational delay or latency is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector passes under the read/write head.

Q.89 *Define random access memory.* **Dec.-10**

Ans. : The memory in which it is possible to access any memory location at random is called random access memory.

Q.90 *What are the advantages of using virtual memory ?* **Dec.-10, May-12**

Ans. : The virtual memory allows the main memory appear larger to user programs than it really is.

Q.91 *How is the average access time of two level memory systems computed ?* **May-10**

Ans. : Refer section 4.8.

Q.92 *What will be the width of address and data buses for a 512 K × 8 memory chip ?* **Dec.-07**

Ans. : Data bus = 8 bit give Address bus = 19 bit.

Q.93 *List the factors that determine the storage device performance.* **May-07**

Ans. : Refer section 4.8.

Q.94 *State the two methods of dynamic memory allocation.*

Ans. : Non preemptive and preemptive.

Q.95 *What is thrashing ?*

Ans. : It is important to select the best suitable memory block for the allocation of particular program, because efficient memory allocation algorithm result in a high hit ratio and a low average access time. If the hit ratio is low, an excessive amount of swapping between memory levels occurs, a phenomenon known as **thrashing**.

Q.96 *What is memory allocation ?*

Ans. : The placement of blocks of information in a memory system is called **memory allocation**.

Q.97 *What is compaction ?*

Ans. : The method by which the existing blocks in the memory are reallocated to make a room for incoming blocks and thus use available memory space more efficiently is known as compaction.

Q.98 *Compare nonpreemptive allocation and preemptive allocation.*

Ans. :

Sr. No.	Preemptive allocation	Non-preemptive allocation
1.	Can make efficient use of memory.	Cannot make efficient use of memory.
2.	Existing blocks are reallocated by method called compaction.	Existing blocks are not reallocated.
3.	Extra-time is required for compaction.	No time is wasted in compaction.

Q.99 *What is meant by internal and external fragmentation ?*

Ans. : Unusable space within a partially filled page frame is called internal fragmentation and unusable space between occupied regions is called external fragmentation.

Q.100 *What is DRO and NDRO ?* **Dec.-11**

Ans. : In some memories reading the memory destroys the stored information. Such phenomenon is called Destructive ReadOut (DRO). Memories in which reading does not affect the stored data have NonDestructive ReadOut (NDRO).

Q.101 *State the three characteristics of memories that destroy information.*

Ans. : The three characteristics of memories that destroy information are : Destructive readout, dynamic storage and volatility.

Q.102 *State the major variables on which hit-rate depends on.*

Ans. : The major variables on which hit-rate depends on are :

- The address streams encountered.
- The average block size.
- The capacity of M1.
- The replacement policy.

Q.103 *What is trace cache ?* **Dec.-12**

Ans. : It is an instruction cache in a microprocessor that stores dynamic instruction sequences after they have been fetched and executed in order to follow the instructions at subsequent times without needing to return to the regular cache or the memory for the same instruction sequence. An advantage of the trace cache is it reduces the required fetch bandwidth on the processing pipeline.

❑❑❑

Notes

5 System Organization

Syllabus

Communication methods, Buses, Bus control, Bus interfacing, Bus arbitration, IO and system control, IO interface circuits, Handshaking, DMA and interrupts, Vectored interrupts, PCI interrupts, Pipeline interrupts, IOP organization, Operation systems, Multiprocessors, Fault tolerance.

Contents

5.1 Introduction

5.2 Communication Methods **May-06, 08, 09, 10, 11,**
. . . . . **Dec.-06, 07, 11, 12,** · · · · · · Marks 10

5.3 I/O and System Control **May-06, 07,08, 09, 10, 12**
. . . . . **Dec.-06,07, 08, 10, 11, 12,** · Marks 16

5.4 Comparison between Programmed I/O and Interrupt Driven I/O . . . . . **May-07, 08, 09,** · · · · · · · · · · Marks 16

5.5 Direct Memory Access (DMA) **May-07, 08, 09, 12,**
. . . . . **Dec.-03,06,07,08,10,11,12,** Marks 16

5.6 Interface Circuit **May-12**

5.7 I/O Channel

5.8 I/O Processor . . . . . **Dec.-03,** · · · · · · · · · · · · · · · · Marks 8

5.9 Operating Systems

5.10 Multiprocessors . . . . . **May-11,** · · · · · · · · · · · · · · · · · Marks 6

5.11 Fault Tolerance . . . . . **Dec.-03, 11,** · · · · · · · · · · · · · Marks 8

Two Marks Questions with Answers

5.1 Introduction

We know that, computers and their major components are interconnected. These interconnections are implemented at the processor or system level. Separate procedures are used for internal and external communication using them. This chapter discusses various ways of communication methods and the design of input/output systems required for communication. The later part of this chapter gives information about multiprocessor systems to achieve high performance, fault tolerance or both.

5.2 Communication Methods

May-06, 08, 09, 10, 11, Dec.-06, 07, 11, 12

The study of computer communication involves the study of a computer's internal or local communication methods and the study of external, long distance communication methods. The communication within a single computer system is also known as **intrasystem** communication, and the communication over long distances is also known as **intersystem communication**. The intrasystem communication is primarily implemented by groups of electrical wires called buses, which support parallel data transmission. On the other hand, the intersystem communication is implemented by a variety of physical media, including electrical cables, optical fibers, and wireless (radio) links. Here, serial (bit by bit) data transmission is preferred rather than parallel data transmission. The parallel data transmission is impractical for long distance communication due to the increase in cost of cabling.

5.2.1 Buses

A group of wires connecting two or more devices and providing communication path way is called bus. A bus that connects major computer components/modules (CPU, memory, I/O) is called a system bus. The system bus is a set of conductors that connects the CPU, memory and I/O modules. Usually, the system bus is separated into three functional groups :

- Data Bus
- Address Bus
- Control Bus

1. Data Bus : The data bus consists of 8, 16, 32 or more parallel signal lines. These lines are used to send data to memory and output ports, and to receive data from memory and input port. Therefore, data bus lines are bi-directional. This means that CPU can read data on these lines from memory or from a port, as well as send data out on these lines to a memory location or to a port. The data bus is connected in parallel to all peripherals. The communication between peripheral and CPU is activated by giving output enable pulse to the peripheral. Outputs of peripherals are floated when they are not in use.

2. Address Bus : It is an unidirectional bus. The address bus consists of 16, 20, 24 or more parallel signal lines. On these lines the CPU sends out the address of the memory location or I/O port that is to be written to or read from. Here, the communication is one way, the address is send from CPU to memory and I/O port and hence these lines are unidirectional.

3. Control Bus : The control lines regulate the activity on the bus. The CPU sends signals on the control bus to enable the outputs of addressed memory devices or port devices.

Fig. 5.2.1 shows bus interconnection scheme.

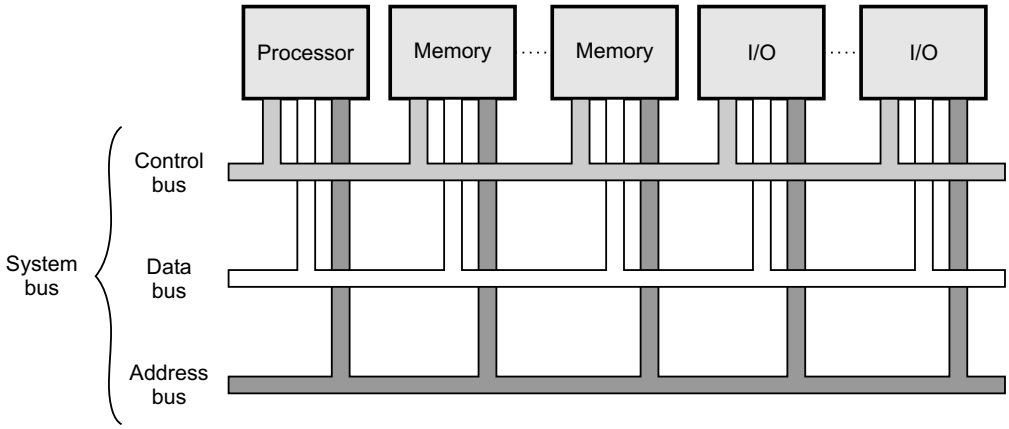


Fig. 5.2.1 Bus interconnection scheme

5.2.1.1 Single Bus Structure

Another way to represent the same bus connection scheme is shown in Fig. 5.2.2. Here, address bus, data bus and control bus are shown by single bus called system bus. Hence such interconnection bus structure is called **single bus structure**.

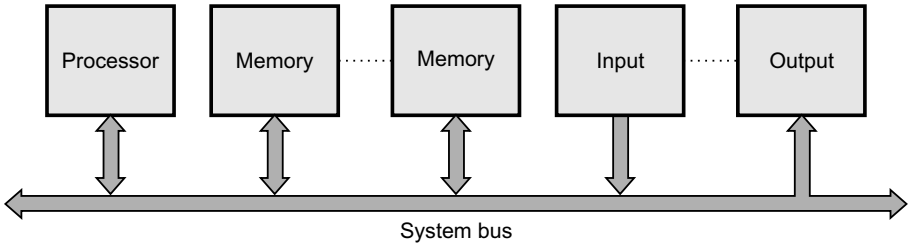


Fig. 5.2.2 Single bus structure

In a single bus structure all units are connected to common bus called system bus. However, with single bus only two units can communicate with each other at a time. The bus control lines are used to arbitrate multiple requests for use of the bus. The main advantage of single bus structure is its low cost and its flexibility for attaching peripheral devices.

The complexity of bus control logic depends on the amount of translation needed between the system bus and CPU, the timing requirements, whether or not interrupt management is included and the size of the overall system. For a small system, control signals of the CPU could be used directly to reduce handshaking logic. Also, drivers and receivers would not be needed for the data and address lines. But large systems with several interfaces would need bus driver and receiver circuits connected to the bus in order to maintain adequate signal quality. In most of the processors, multiplexed address and data buses are used to reduce the number of pins. During first part of bus cycle, address is present on this bus. Afterwards, the same bus is used for data transfer purpose. So latches are required to hold the address sent by the CPU initially. Interrupt priority management is optional in a system. It is not required in systems which use software priority management. The complex system includes hardware for managing the I/O interrupts to increase efficiency of a system. Many manufacturers have made priority management devices. Programmable interrupt controller (PIC) is the IC designed to fulfil the same task.

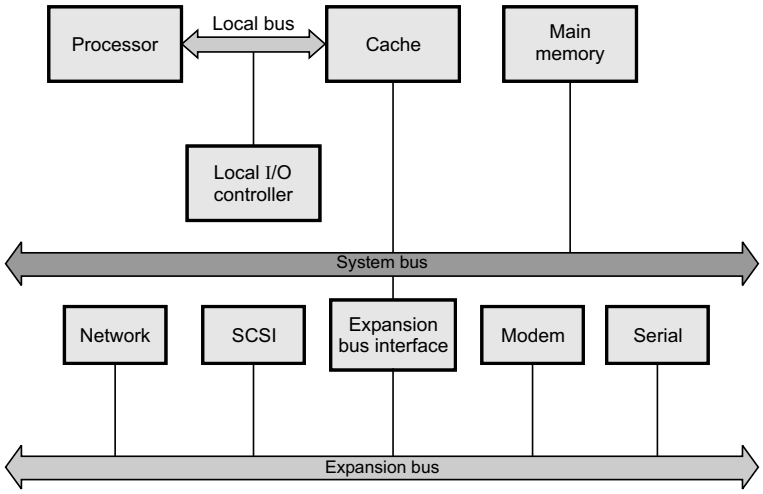
5.2.1.2 Multiple Bus Structures

The performance of computer system suffers when large number of devices are connected to the bus. This is because of the two major reasons :

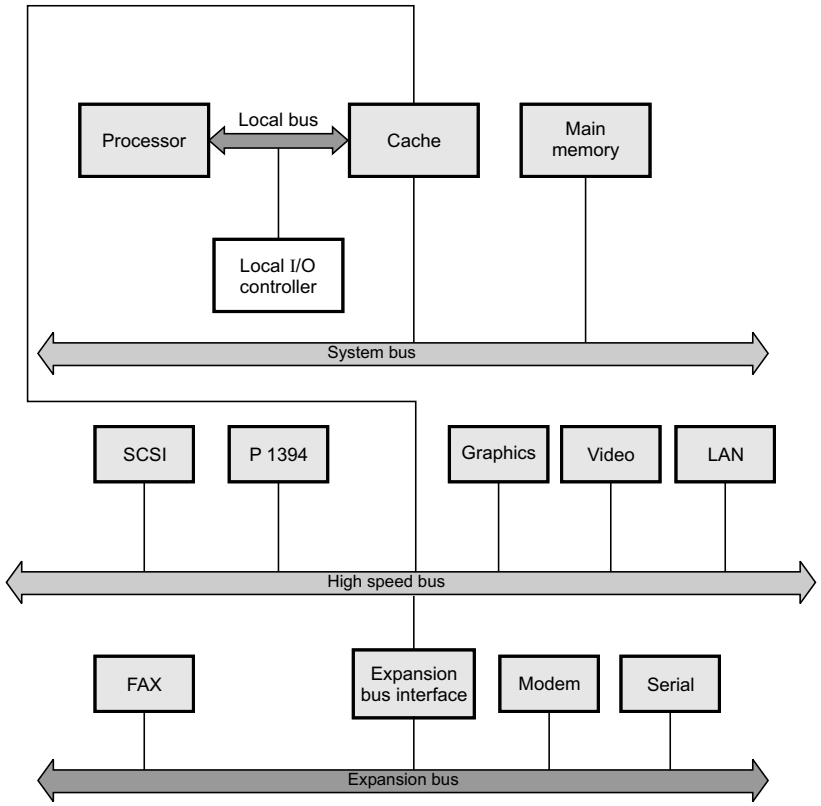
1. When more devices are connected to the common bus we need to share the bus amongst these devices. The sharing mechanism co-ordinates the use of bus to different devices. This co-ordination requires finite time called **propagation delay**. When control of the bus passes from one device to another frequently, these propagation delays are noticeable and affect the performance of computer system.
2. When the aggregate data transfer demand approaches the capacity of the bus, the bus may become a bottleneck. In such situations we have to increase the data rate of the bus or we have to use wider bus.

Now-a-days the data transfer rates for video controllers and network interfaces are growing rapidly. The need of high speed shared bus is impractical to satisfy with a single bus. Thus, most computer systems use the multiple buses. These buses have the hierarchical structure.

Fig. 5.2.3 shows two bus configurations. The traditional bus connection uses three buses : local bus, system bus and expanded bus. The high speed bus configuration uses high-speed bus along with the three buses used in the traditional bus connection. Here, cache controller is connected to high-speed bus. This bus supports connection to high-speed LANs, such as Fiber Distributed Data Interface (FDDI), video and graphics workstation controllers, as well as interface controllers to local peripheral buses including SCSI and P1394.



(a) Traditional bus configuration



(b) High-speed bus configuration

Fig. 5.2.3

5.2.1.3 Bus Design Parameters

In the previous section we have seen various buses and their functions. To implement these buses in a computer system we have to consider following parameters :

- Type of Bus : Dedicated or multiplexed
- Method of Arbitration : Centralized or distributed
- Timing : Synchronous or asynchronous
- Bus Width : Address or Data
- Data Transfer Type : Read, write, read modify write, read-after write or block.

Bus Types

When a particular bus is permanently assigned to one function or to a physical subset of computer component, it is called **dedicated bus**. As an example of functional dedication is the use of separate dedicated address and data lines. Physical dedication refers to the use of separate buses of I/O and memory modules. The physical dedication gives high throughout, because there is less bus contention. A disadvantage is the increased size and cost.

When bus is used for more than one function at different time zones, it is called **multiplexed bus**. An example of multiplexed bus is the use of same bus for address and data. At the beginning of data transfer, the address is placed on the bus along with the address latch enable signal (ALE). At this point module has sufficient time to latch the address with latch ICs. The address is then removed from the bus and the same bus is used for data transfer.

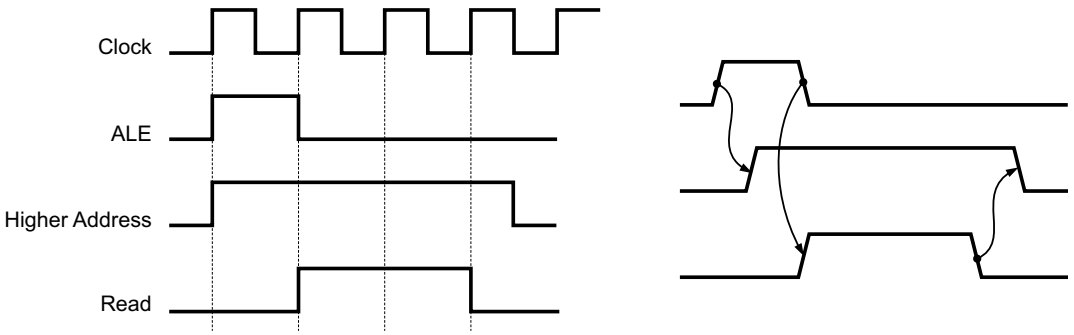
Method of Arbitration

In a computer system there may be more than one bus masters such as processor, DMA controller, etc. They share the system bus. In a centralized approach, a single hardware device, referred to as a bus controller or arbiter, is responsible for allocating time on the bus for bus masters. In the distributed approach each bus master contains access control logic and they act together to share the bus.

Bus Timings

Bus timing refers to the way in which events are coordinated on the bus. In synchronous timing, the occurrence of events on the bus is determined by the clock whereas in the asynchronous timing, the occurrence of events on the bus is determined by the previous events on the bus.

Fig. 5.2.4 shows synchronous and asynchronous timings.



(a) Synchronous timings (b) Asynchronous timings

Fig. 5.2.4 Bus timings

Bus Width

Bus width is nothing but the number of lines which are grouped to form a bus. Data bus width decides the number of bits transferred at one time, whereas address bus decides the range of locations that can be accessed.

Data Transfer Type

As mentioned earlier, there are various data transfer types such as read, write, read-modify write, read after write and block.

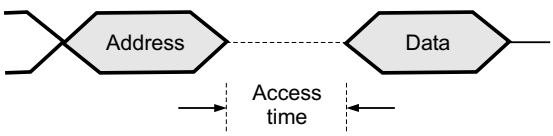


Fig. 5.2.5 Read data transfer

Read : In this operation address is put on the address bus and after sufficient access time data is available on the data bus. Fig. 5.2.5 shows read data transfer for multiplexed address and data bus.

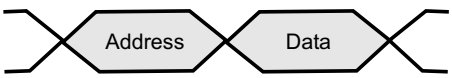


Fig. 5.2.6 Write data transfer

Write : In this operation address is put on the multiplexed bus and then immediately after latching period data is put on the multiplexed bus, as shown in the Fig. 5.2.6.

Read Modify Write : In this operation, read data transfer is followed by write data transfer at the same address.

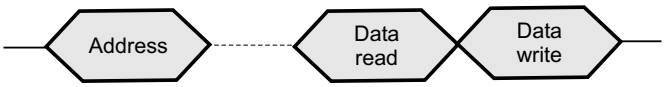


Fig. 5.2.7 Read-Modify-Write data transfer

Read-After Write : In this operation, write data transfer is followed by read data transfer at the same address.



Fig. 5.2.8 Read-after-Write data transfer

Block : In this operation, the address is put on the address bus and then the n number of data transfer cycles are carried out.

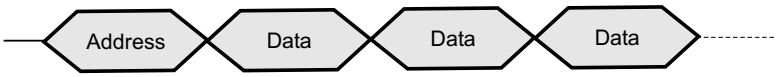


Fig. 5.2.9 Block data transfer

5.2.2 Bus Control

In this section we are going to see the methods to establish and control intrasystem communication via a **shared** or **multiplexed bus**. In shared bus there are two key issues : 1. To get access to the bus and 2. The timing of transfers over the bus.

Let us refer Fig. 5.2.1 as a general structure of the system and its interconnections. In which one particular unit acts as the bus master and supervises the use of the bus by the other units, the bus slaves. In most of the cases the CPU is the bus master, while the memory and I/O interface circuits are slaves. However, IO controllers also serve as bus masters. Only a bus master can initiate data transfer, although slaves can request for data transfer. Once the data transfer is initiated, both master and slave participate equally in the data-transfer process.

The data transfer can be carried out in two ways :

- Synchronous
- Asynchronous

5.2.2.1 Synchronous Input/Output Transfer

In synchronous bus, all devices derive timing information from a common clock signal. The Fig. 5.2.10 shows the timing diagram for synchronous input/output transfer. At time t_0, the processor places the device address on the address lines of system bus and sets the control lines to perform the input operation. During time $t_0 - t_1$, addressed device gets the address and it recognizes that an input operation is requested. At time t_1, address device places its data on the data bus. At the end of bus cycle, that, is at time t_2, the processor reads the data lines and loads the data from data bus into its input buffer.

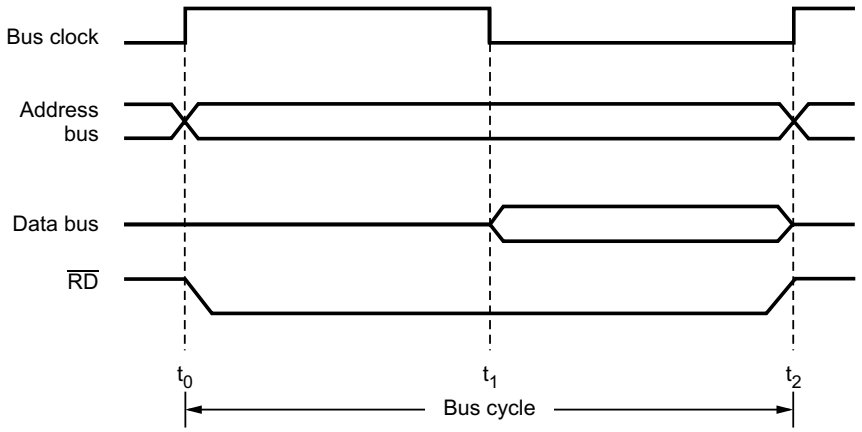


Fig. 5.2.10 Timing diagram for synchronous input transfer

The timing diagram for an output operation is shown in the Fig. 5.2.11. Here, processor places the output data on the data lines when it transmits the address and control line informations. At time t_1, the addressed device reads the data lines and loads the data into its data buffer. The processor simply assumes that, at t_2, the output data have been received by the I/O device and the cycle ends.

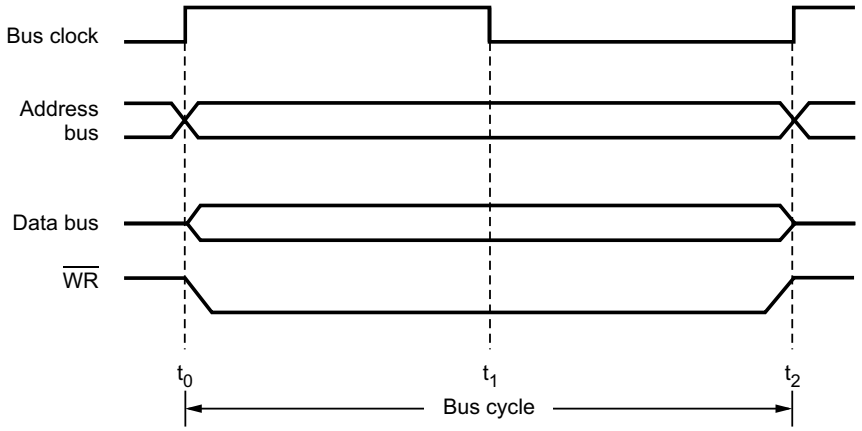


Fig. 5.2.11 Timing diagram for synchronous output transfer

In synchronous system, the clock pulse width should be chosen such that it is greater than the maximum propagation delay between the processor and any of the devices connected to the bus. During read operation, the clock pulse width must be large enough to allow all devices to decode the address and control signals so that the addressed device can be ready to respond at time t_1.

The timing diagrams shown in Fig. 5.2.10 and Fig. 5.2.11 do not consider the propagation delay so they represent ideal timings for data transfer that takes place on

lines. In practice the propagation delay, delays the timings at which signals actually change state. This is illustrated in Fig. 5.2.12. The Fig. 5.2.12 shows the signals as seen by the master and the other as seen by the slave.

The master sends address signals on the rising edge at the beginning of t_0. But due to the delay in the bus driver circuit address signals appear on the address bus after time delay of t_{AM}. After some more delay, at t_{AS}, the address signals reach the slave. The slave decodes the address and at t_1 sends the requested data. Here again, the data signals do not appear on the data bus until t_{DS}. Data signals reach to the master at time t_{DM}. At t_2, the master reads and loads the data into its input buffer. The time $t_2 - t_{DM}$ is known as data setup time for master's input buffer. The data must continue to be valid after t_2 for a period equal to the hold time of the master's input buffer.

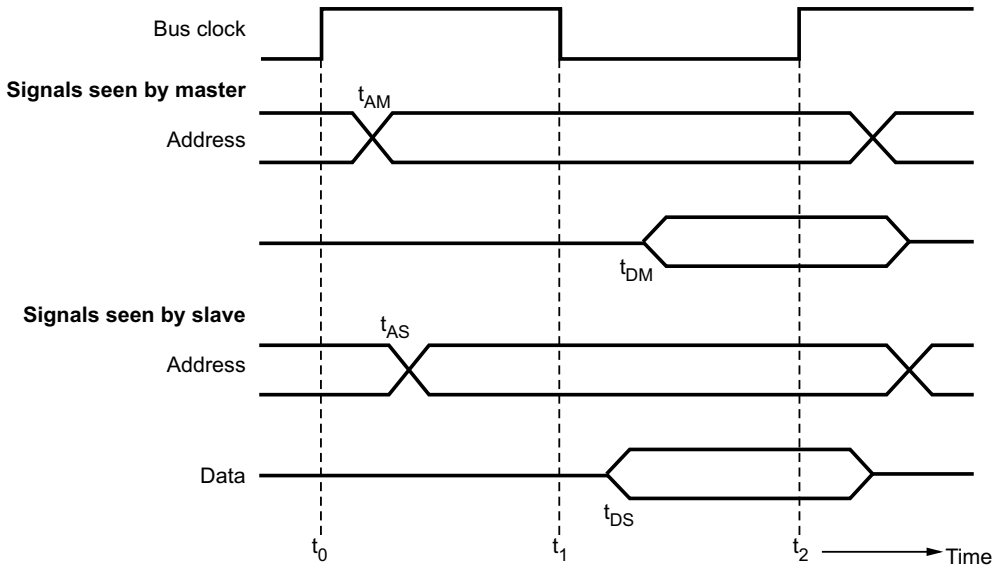


Fig. 5.2.12 Timing diagram for synchronous input transfer considering propagation delays

Limitations of above discussed scheme :

- Because a transfer has to be completed within one clock cycle, the clock period, $t_2 - t_0$, must be selected to accommodate the longest delays on the bus and the slowest device interface. This forces all devices to operate at the speed of the slowest device.
- There is no way determine whether the slave is responding to data transfer or not.

To overcome above mentioned limitations most of the buses includes control signals that represent a response from the device and use high frequency clock signal such that a complete data transfer cycle would span several clock cycles. Then, the number of clock cycles involved can vary from one device to another and operating speed is not

decided by the slowest device. The Fig. 5.2.13 shows the timing diagram for modified synchronous input data transfer with multiple clock cycles.

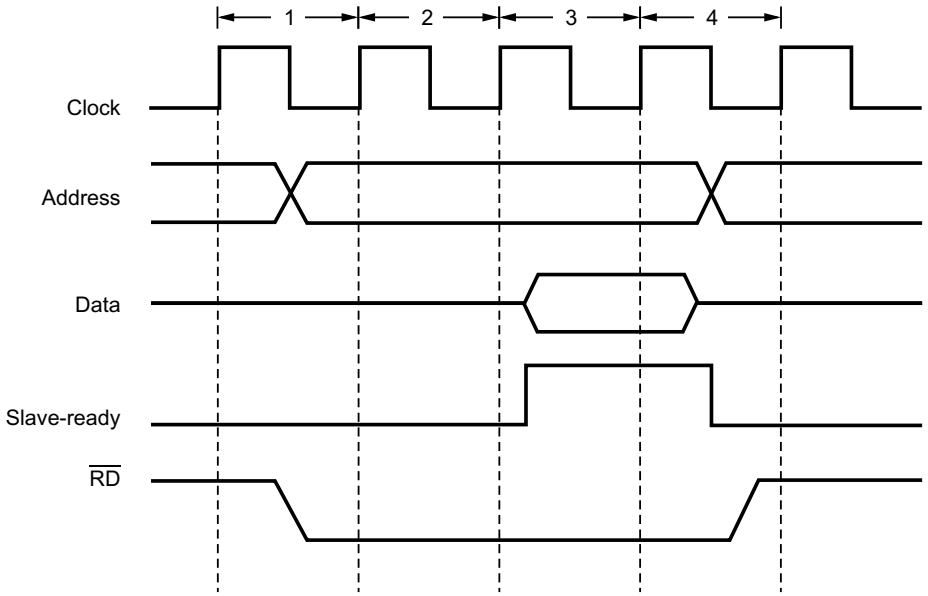


Fig. 5.2.13 Modified synchronous input data transfer with multiple clock cycles

5.2.2.2 Asynchronous Input/Output Transfer

In asynchronous bus, the common clock is eliminated and data transfer on the system bus is achieved by the use of a hand shake between the processor and the device being addressed. Here, the clock line is replaced by two control signals Ready and Accept.

For input operation, processor places the address and sets the control lines to perform the input operation at t_0. It then indicates at t_1 to the addressed device that it is ready to accept data by activating the Ready line. When the addressed device receives the Ready signal, it places the data on the data bus at t_2 and then informs the processor that it has done so by activating the Accept line. The processor waits for accept line. Upon activation of Accept line processor reads data from the data bus and loads it into the input buffer the timing diagram for input operation is as shown in Fig. 5.2.14 At t_4 processor removes the address and controls information from bus. When the device interface receives the 1 to 0 transition of master ready signal, it removes the data and slave-ready signal from the bus. This completes the input data transfer.

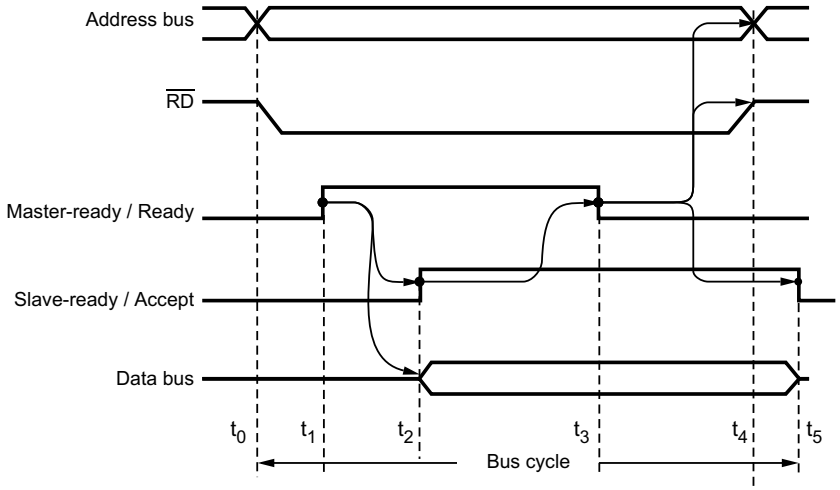


Fig. 5.2.14 Timing diagram for asynchronous input transfer

For output operation, processor places the address, data and sets the control lines to perform the output operation. It then indicates to the addressed device that it has send data by activating Ready signal. The addressed device reads the data into its input buffer when it receives Ready signal. It then indicates that it has done so by activating the Accept signal. The timing diagram for output operation is as shown in Fig. 5.2.15.

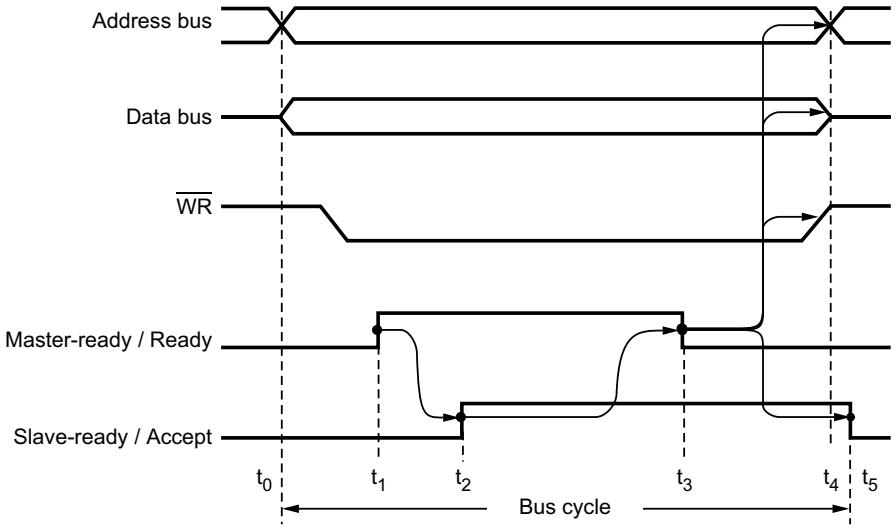


Fig. 5.2.15 Timing diagram for asynchronous output transfer

Two commonly used methods are used to transfer data asynchronously between two independent units are :

- Strobe
- Handshake

5.2.2.3 Strobe Control

In the strobe control asynchronous data transfer, a single control line (strobe) is used to indicate to the other unit when the transfer has to occur. The strobe signal may be activated by either the source or the destination unit. The Fig. 5.2.16 shows a source-initiated data transfer. In this the strobe signal informs the destination unit that the valid data is available on the data bus from the source unit.

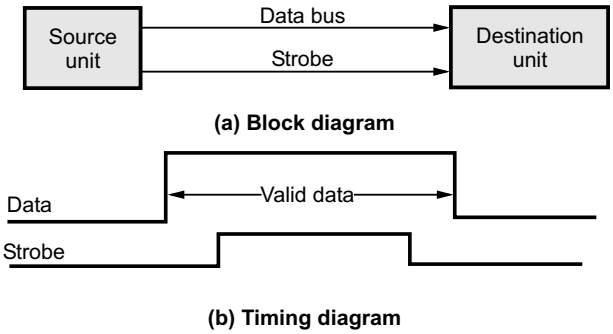


Fig. 5.2.16 Source initiated strobe for data transfer

As shown in the Fig. 5.2.16 (b), the source unit first places the data on the data bus. After a data setup time (time require to settle data to steady state value on the data bus), the source activates the strobe pulse. The strobe signal and data on the data bus remain in active state for a sufficient time to allow the destination unit to receive the data. After that strobe is inactivated and the source removes the data from the data bus. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal register. The Fig. 5.2.17 shows a destination initiated data transfer. In this case, the destination unit activates the strobe pulse, informing the source to provide the data. The source unit in response to strobe places the valid data on the data bus. The data is made available for enough period to allow the destination unit to receive it. The falling edge of the strobe pulse can be used again to trigger a destination register.

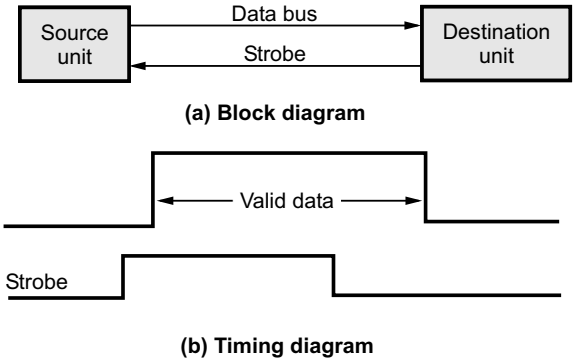
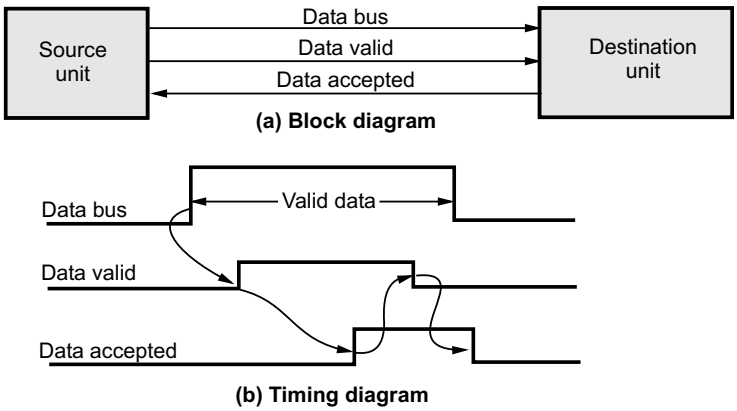


Fig. 5.2.17 Destination initiated strobe for data transfer

5.2.2.4 Handshaking

The disadvantage of strobe method is that the source unit that initiates the transfer has no idea whether the destination unit has actually accepted the data sent by it. Similarly, a destination unit that initiates the transfer has no idea whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing one more signal called **acknowledge signal**. The acknowledge signal provides a reply to the unit that initiates the data transfer.



The Fig. 5.2.18 shows the data transfer using handshake signals when initiated by source unit. Here, the source unit initiates the data transfer by placing the data on the bus and enabling its **data valid signal**. In response to this, the destination unit accepts the data and it sends **data accepted** (Acknowledge) signal to the source unit. The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial stage.

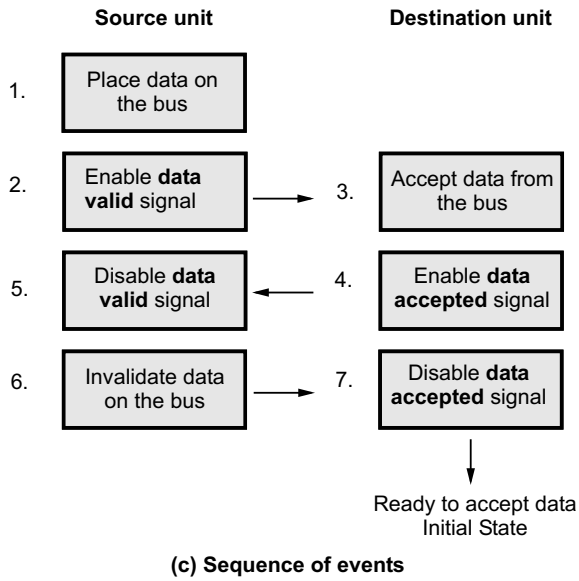

Source unit
Destination unit
1. Place data on the bus
2. Enable data valid signal
3. Accept data from the bus
4. Enable data accepted signal
5. Disable data valid signal
6. Invalidate data on the bus
7. Disable data accepted signal
Ready to accept data
Initial State
(c) Sequence of events

Fig. 5.2.18 Source initiated data transfer using handshaking

The Fig. 5.2.19 shows the data transfer using handshaking signals when initiated by destination unit. Here, the destination unit activates **ready for data** signal when it is ready to accept data from source unit. In response to that, the source unit places the data on the bus and initiates the **data valid** signal. The destination unit then accepts the data from the data bus and disables the **ready for data** signal. Then, the source unit disables **data valid** signal and invalidates data on the bus. As a result, the system goes into its initial stage.

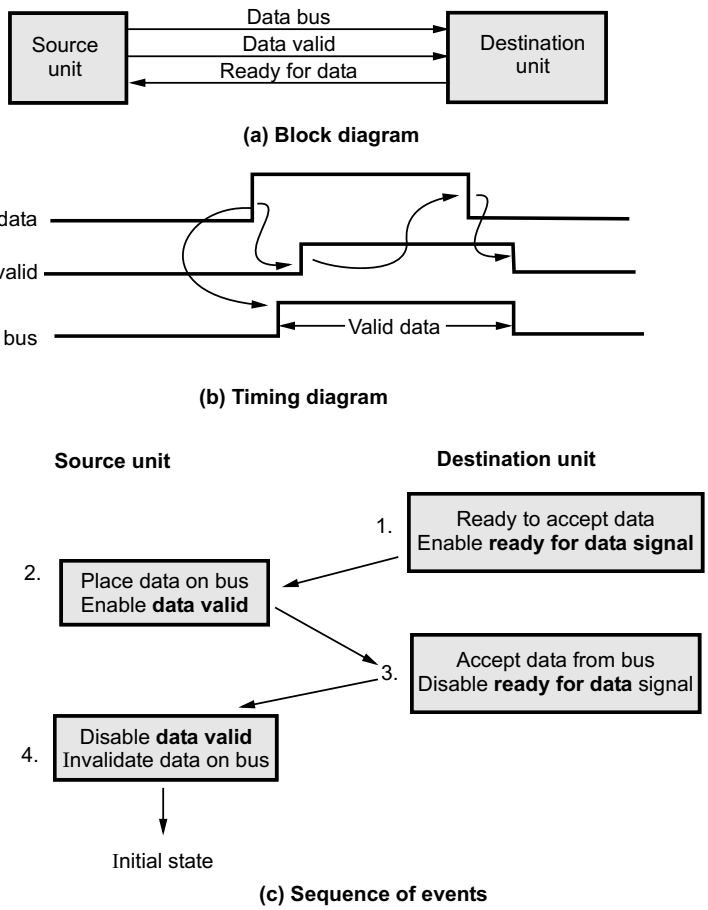


Fig. 5.2.19 Destination initiated data transfer using handshaking

Advantages of handshaking scheme

1. The handshaking scheme provides high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units.
2. The scheme can use **timeout** signal to detect any problem in the completion of data transfer. For example, if one unit enables its handshaking control signal and if the return handshaking signal does not respond within a given time period, the unit detects an error and activates **timeout** signal to interrupt processor. In response, processor executes a service routine that takes appropriate error recovery action.

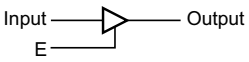
5.2.3 Bus Interfacing

The bus is connected to transfer data and control signal between various devices. The bus being single path for data communication, it should have very large fan-in and

fan-out. Thus, buffer circuits called bus drivers and receivers are needed to transfer signals to and from the bus, respectively.

A special tri-state TTL logic is often used in bus design. It is called tristate TTL because it allows three possible output stages : HIGH, LOW and high impedance. HIGH state represents binary 1 value and LOW state represents binary 0 value. On the other hand, the high impedance state, Z, denotes the state of a line that is electrically disconnected from all voltage sources, that is, an open-circuited or floating line.

Fig. 5.2.20 shows logic symbol and truth table for tristate buffer, which serves as a bus-line driver. The tristate buffer has a special input line E called output enable, which when set to 0 disables the output line making it open circuited i.e. output = Z. When E = 1, the circuit becomes an ordinary noninverting buffer with output = input.



(a) Logic symbol

Input	Enable	Output
X	0	Z
0	1	0
1	1	1

(b) Truth table

Fig. 5.2.20 Tri-state buffer

As shown in the Fig. 5.2.21 there are two output devices which can give data to the one input device. These three devices are connected together with a common data bus. The common data bus is shared by the two output devices. When CONTROL signal is LOW device 1 is enabled and device 2 is disabled. As a result the data (8-bit) from device 1 is available on the data bus, as shown in the Fig. 5.2.21 (a). When CONTROL

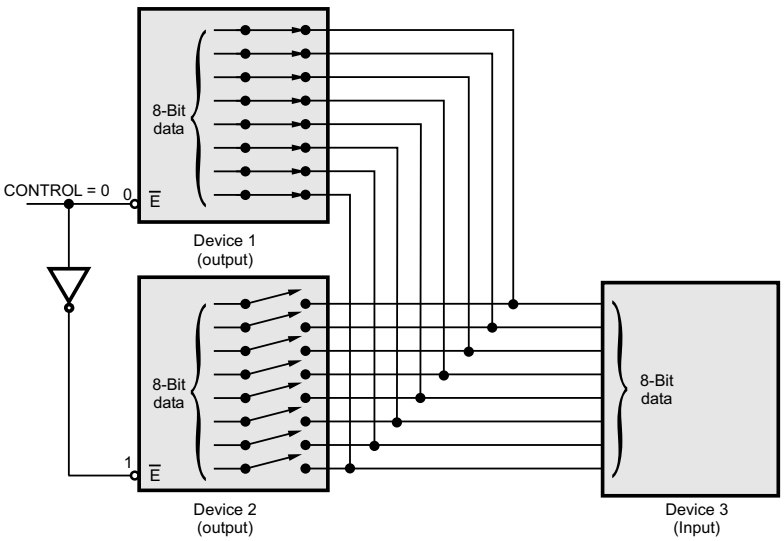


Fig. 5.2.21 (a)

signal is HIGH device 1 is disabled and device 2 is enabled. In this case the data from the device 2 is available on the data bus as shown in the Fig. 5.2.19 (b).

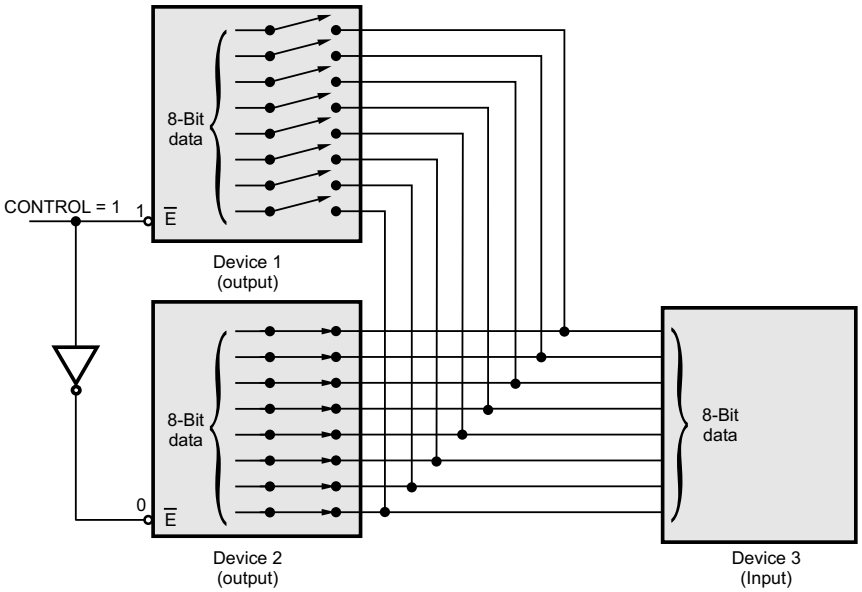


Fig. 5.2.21 (b)

As said earlier bus is a common connection between number of devices. When devices are more, required current sourcing and sinking capabilities may exceed than those provided by standard TTL. For such requirements buffer ICs are used. They can source and sink more current than a standard TTL. The buffer ICs used for increasing sinking and sourcing capabilities of bus are called bus drivers.

Any logic circuit that is called a buffer, or a driver is designed to have a greater output current and /or voltage capability than an ordinary logic circuit. Buffer/driver ICs are available with totem-pole outputs and with open-collector outputs.

A typical TTL bus driver contains eight inverters, or eight noninverting drivers, all with tristate outputs and enable signal. Many have more than one enable input and/or built-in gates that can be used to perform logic operations on the enable inputs (s). Many versions have separate enable inputs that can be of four drivers each, as shown in the Fig. 5.2.22 (a). Bus transceivers are two-directional bus drivers : They can be used to pass data in one direction while the other direction is in high impedance state and vice-versa. They are also known as bidirectional bus drivers. Fig. 5.2.22 (b) shows the logic diagram for bidirectional bus driver.

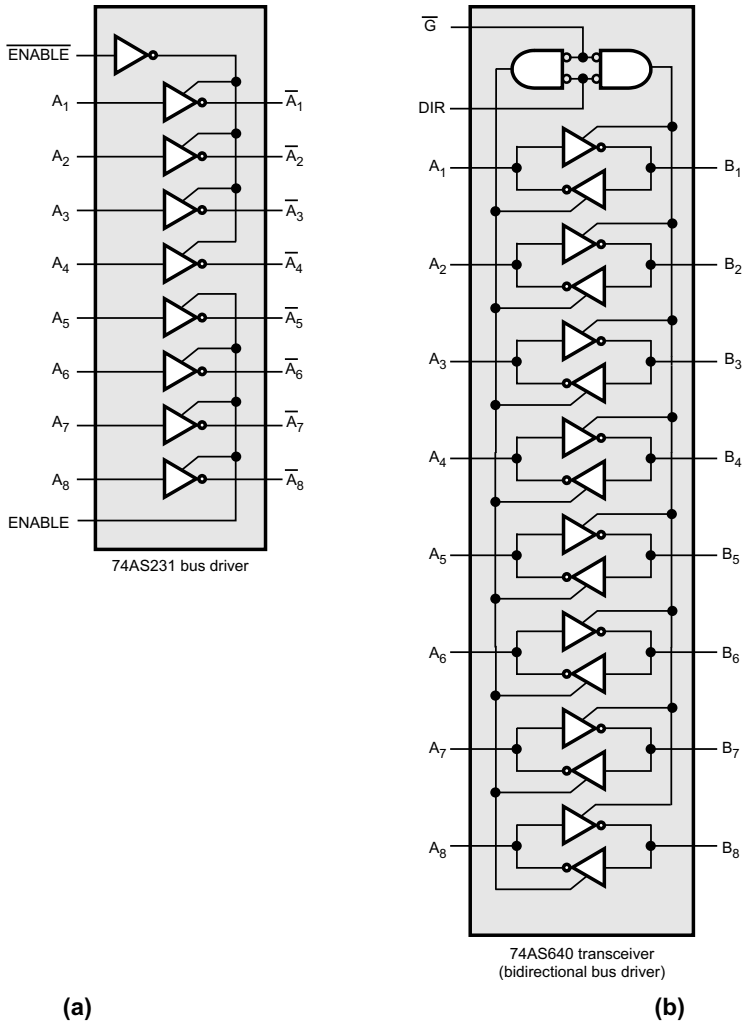


Fig. 5.2.22 A typical tristate bus driver and transceiver

5.2.4 Bus Arbitration

The device that is allowed to initiate data transfers on the bus at any given time is called the **bus master**. In a computer system there may be more than one bus master such as processor, DMA controller etc. They share the system bus. When current master relinquishes control of the bus, another bus master can acquire the control of the bus. Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of bus master is usually done on the priority basis. There are two approaches to bus arbitration : centralized and distributed.

5.2.4.1 Centralized Arbitration

In centralized bus arbitration, a single bus arbiter performs the required arbitration. The bus arbiter may be the processor or a separate controller connected to the bus. There are three different arbitration schemes that use the centralized bus arbitration approach. These schemes are :

a. Daisy chaining

b. Polling method

c. Independent request

a) Daisy chaining : The system connections for Daisy chaining method are shown in Fig. 5.2.23. It is a simple and cheaper method. All masters make use of the same line for bus request. In response to a bus request the controller sends a bus grant if the bus is free. The bus grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This master blocks the propagation of the bus grant signal, activates the busy line and gains control of the bus. Therefore any other requesting module will not receive the grant signal and hence cannot get the bus access.

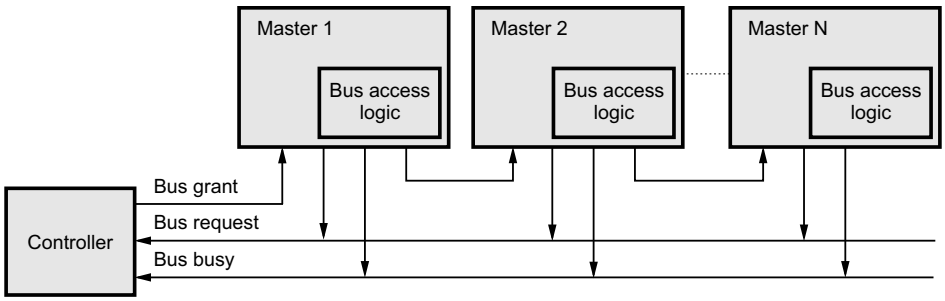


Fig. 5.2.23 Daisy chaining method

Advantages

1. It is a simple and cheaper method.
2. It requires the least number of lines and this number is independent of the number of masters in the system.

Disadvantages

1. The propagation delay of bus grant signal is proportional to the number of masters in the system. This makes arbitration time slow and hence limits the number of masters in the system.
2. The priority of the master is fixed by its physical location.
3. Failure of any one master causes the whole system to fail.

b) Polling method : The system connections for polling method are shown in Fig. 5.2.24. In this the controller is used to generate the addresses for the masters. Number of address lines required depends on the number of masters connected in the system. For example, if there are 8 masters connected in the system, at least three address lines are required. In response to a bus request controller generates a sequence of master addresses. When the requesting master recognizes its address, it activates the busy line and begins to use the bus.

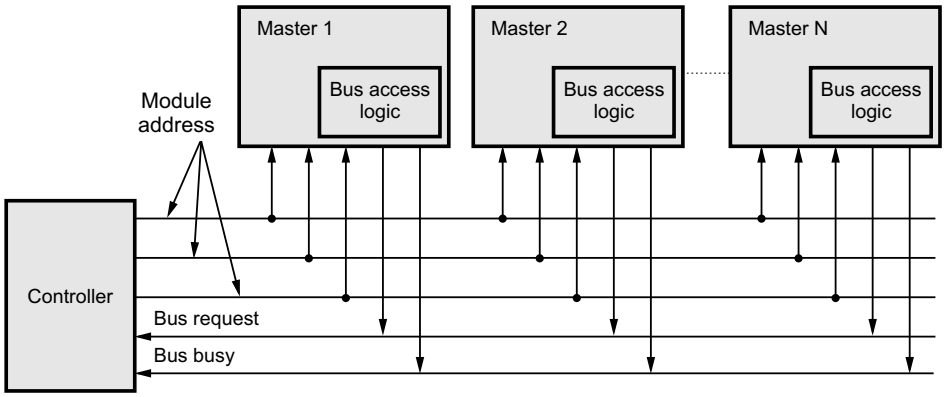


Fig. 5.2.24 Polling method

Advantages

1. The priority can be changed by altering the polling sequence stored in the controller.
2. If the one module fails entire system does not fail.

c) Independent request : The Fig. 5.2.25 shows the system connections for the independent request scheme. In this scheme each master has a separate pair of bus

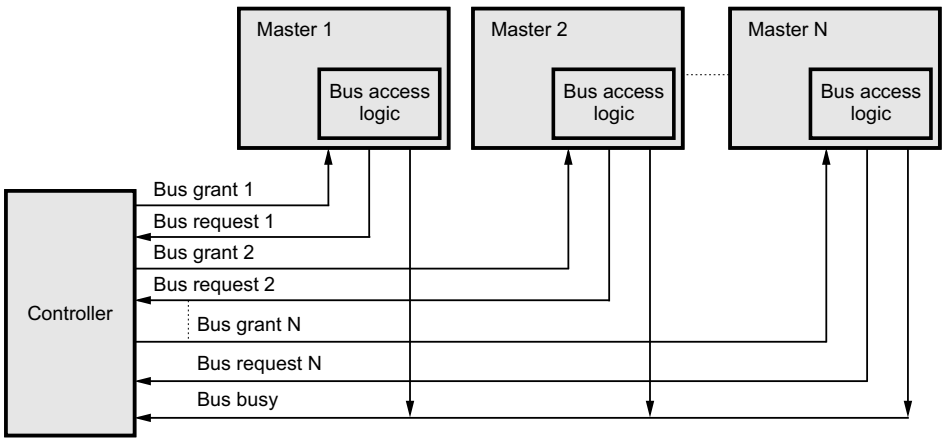


Fig. 5.2.25 Independent request method

request and bus grant lines and each pair has a priority assigned to it. The built in priority decoder within the controller selects the highest priority request and asserts the corresponding bus grant signal.

Advantage

1. Due to separate pairs of bus request and bus grant signals, arbitration is fast and is independent of the number of masters in the system.

Disadvantage

1. It requires more bus request and grant signals (2 × n signals for n modules).

5.2.4.2 Distributed Arbitration

In distributed arbitration, all devices participate in the selection of the next bus master. The Fig. 5.2.26 shows the scheme for distributed arbitration. In this scheme each device on the bus is assigned a 4-bit identification number. The number of bits used for identification number actually depends on the number of devices connected on the bus. When one or more devices request for the control of bus, they assert the Start-Arbitration signal and place their 4-bit ID numbers on arbitration lines, $\overline{ARB0}$ through $\overline{ARB3}$. These four arbitration lines are all open-collector. Therefore, more than one devices can place their 4-bit ID number to indicate that they need the control of bus. If one device puts 1 on the bus line and another device puts 0 on the same bus line, the bus line status will be 0. Device reads the status of all lines through inverter buffers so device reads bus status 0 as logic 1. In this scheme the device having highest ID number has highest priority.

When two or more devices place their ID number on bus lines then it is necessary to identify the highest ID number from the status of bus line. Consider that two devices A and B, having ID numbers 1 and 6, respectively are requesting the use of the bus. Device A puts the bit pattern 0001, and device B puts the bit pattern 0110. With this combination the status of bus-lines will be 1000; however because of inverter buffers code seen by both devices is 0111. Each device compares the code formed on the arbitration lines to its own ID, starting from the most significant bit. If it finds a difference at any bit position, it disables its drives at that bit position and for all lower-order bits. It does so by placing a 0 at the input of these drivers. In our example, device detects a different on line $\overline{ARB2}$ and hence it disables its drives on lines $\overline{ARB2}$, $\overline{ARB1}$ and $\overline{ARB0}$. This causes the code on the arbitration lines to change to 0110. This means that device B has won the race.

The decentralized arbitration offers high reliability because operation of the bus is not dependent on any single device.

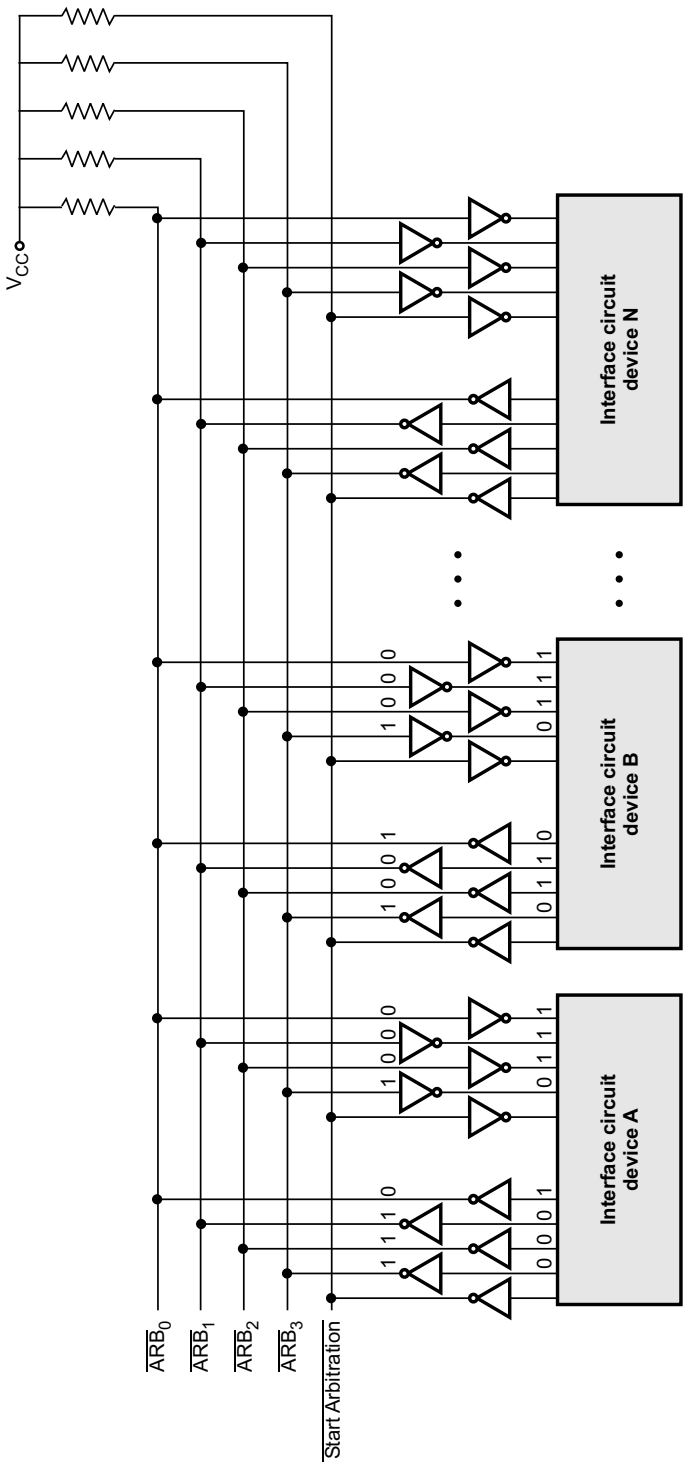


Fig. 5.2.26 A distributed arbitration scheme

Example 5.2.1 *The Fig. 5.2.27 shows a basic arrangement in which the processor contains the bus arbitration circuitry and two DMA controllers are connected as a bus masters using $\overline{BR}$ (Bus request), BG (Bus grant) and $\overline{BBSY}$ (Bus busy) signals.*

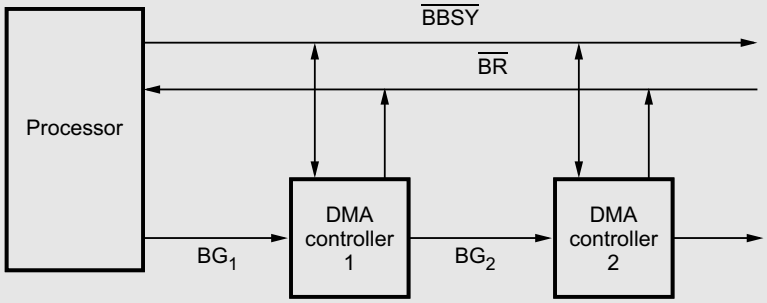


Fig. 5.2.27

Solution : The timing diagram in the Fig. 5.2.28 shows the sequence of events for the devices in the Fig. 5.2.27 as DMA controller 2 requests and acquires but mastership and later releases the bus.

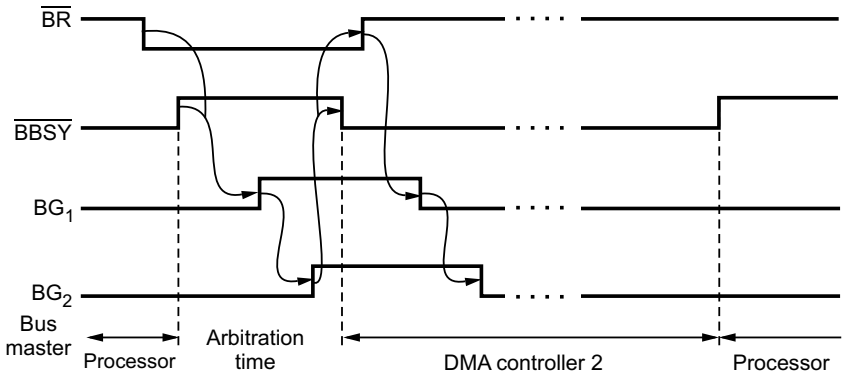


Fig. 5.2.28

Example 5.2.2 *In the arrangement in the Fig. 5.2.29, a device becomes the bus master only when it receives a low-to high transition on its bus grant input. Assume device 1 requests the bus and receives a grant. While it is still using the bus, device 3 asserts it BR output. Draw a timing diagram showing how device 3 becomes the bus master after device 1 releases the bus.*

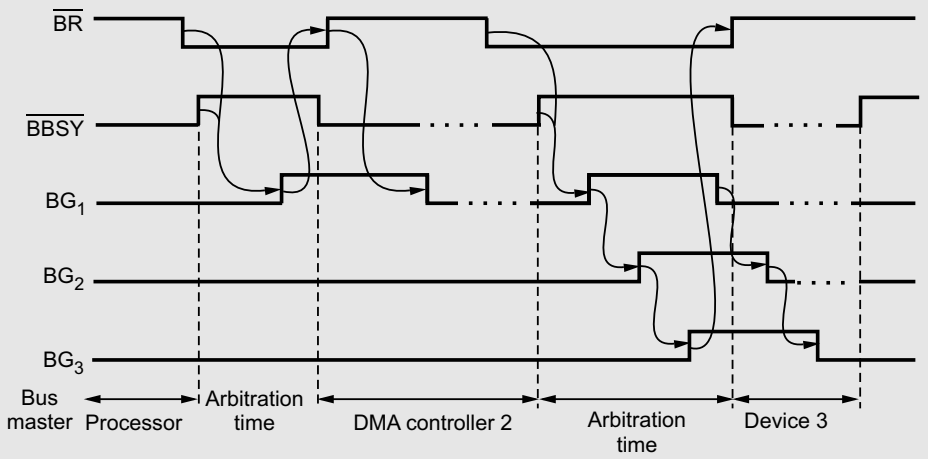


Fig. 5.2.29

Solution : Initially, the processor has a control of bus. After receiving the bus request ($\overline{BR}$) signal low, it checks for bus busy ($\overline{BBSY}$). When bus busy is inactivated it generates the bus grant signal. Since bus request is from device 1 it get the control of bus and make bus busy signal active and do not propagate bus grant signal to further devices. While device 1 is using bus, the bus request from devices 3 do not generate bus grant signals because bus busy line is still active. When device 1 release the bus, bus busy signal becomes inactive and bus grant signal is generated for device 1. It is then propagated to device 2 and device 3 When device 3 gets bus grant signal it becomes master and activates the bus busy signal till it uses the bus.

Example 5.2.3 *Consider the daisy-chain arrangement shown in the Fig. 5.2.30, in which the bus-request signal is fed back directly as the bus grant. Assume that device 3 requests the bus and begins using it. When device 3 is finished, it deactivates BR_3. Assume that the delay from BG_i to $BG_{(i+1)}$ in any device is d. Show that a spurious bus grant pulse will travel downstream from device 3. Estimate the width of this pulse.*

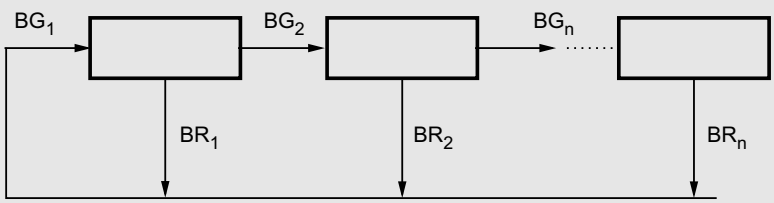


Fig. 5.2.30

Solution : It is important note that in above interface bus busy signal is not used. Instead of bus busy signal bus request signal is deactivated to indicate the release of bus by corresponding device.

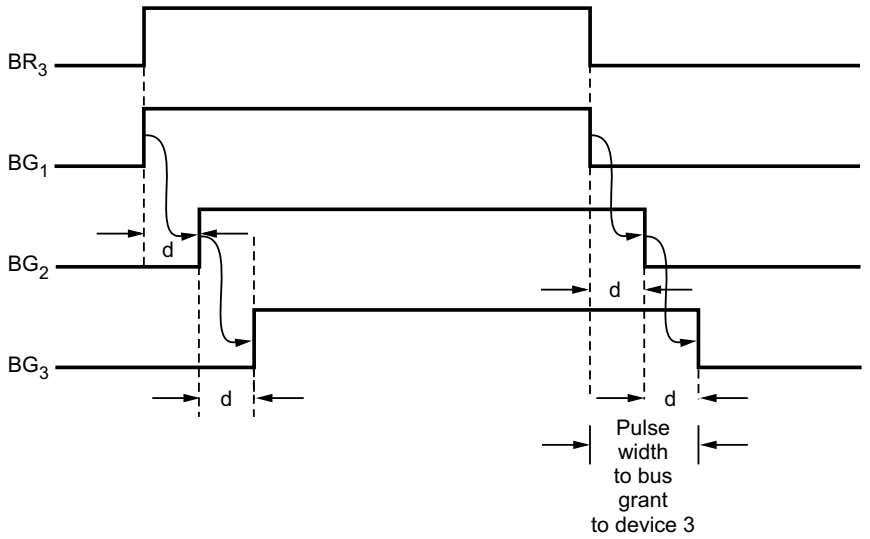


Fig. 5.2.31

Example 5.2.4 *Why does DMA have priority over the CPU when both request a memory transfer ?*

Solution : The DMA controller and the CPU share the bus. Only one of them at a time can have control of the bus, that is, be the bus master. Generally, DMA takes priority over CPU because many I/O devices operate within tight timing parameters. If they detect no activity within a specified period, they timeout and abort I/O process. To avoid device timeouts, the DMA uses memory cycles that would otherwise be used by the CPU. This is called cycle stealing. Fortunately, I/O tends to create bursty traffic on the bus; data is sent in blocks, or clusters. The CPU is granted bus access between bursts.

5.2.5 PCI Bus

In early 1992, Intel formed an another industrial group in relation to be PC bus. The main intension behind the formation of group was to overcome the weaknesses in the ISA and EISA buses. They designed PCI (Peripheral Component Interface) specifications in June 1992, and updated in April 1993.

5.2.5.1 Features

1. It is designed to economically meet the I/O requirements of modern systems. It requires very few chips to implement and support other buses attached to the PCI bus.
2. It bypasses the standard I/O bus, uses the system bus to increase the bus clock speed and take full advantage of the CPU's data path.
3. It has an ability to function with a 64-bit data bus.
4. It has high bandwidth. The information is transferred across the PCI bus at 33 MHz, at the full data width of the CPU. When the bus is used in conjunction with a 32-bit CPU, the bandwidth is 132 Mbytes/sec. It is calculated as follows :

   $$33 \text{ MHz} \times 32\text{-bit} = 1{,}056 \text{ Mbits/sec}$$

   $$1{,}056 \text{ Mbits/sec} \div 8 = 132 \text{ Mbytes/sec}$$

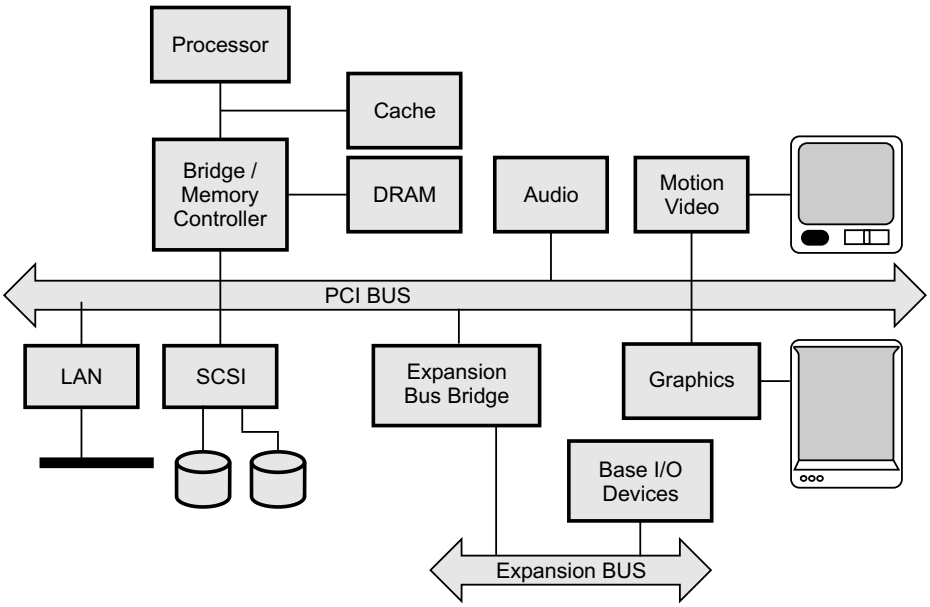
5. PCI bus is designed to support a variety of microprocessor based configurations including both single and multiprocessor systems.
6. The PCI bus can operate concurrently with the processor bus. The CPU can be processing data in a external cache while the PCI bus is busy transferring information between other parts of the system.
7. The PCI bus is processor - independent bus that can function as a mezzanine or peripheral bus.
8. It makes use of synchronous timings and centralized arbitration scheme.
9. It delivers better system performance for high-speed I/O subsystems (e.g. graphic display adapters, network interface controllers, disk controllers and so on).
10. The PCI interface contains a 256 bytes configuration memory which allows the computer to interrogate the PCI interface. This feature allows the system to automatically configure itself for the PCI plug-board and hence it is referred to as plug-and-play.

5.2.5.2 PCI Configurations

As mentioned earlier, PCI bus is designed to support single processor as well as multiprocessor systems. Fig. 5.2.30 (a) shows a typical use of PCI in a single processor system.

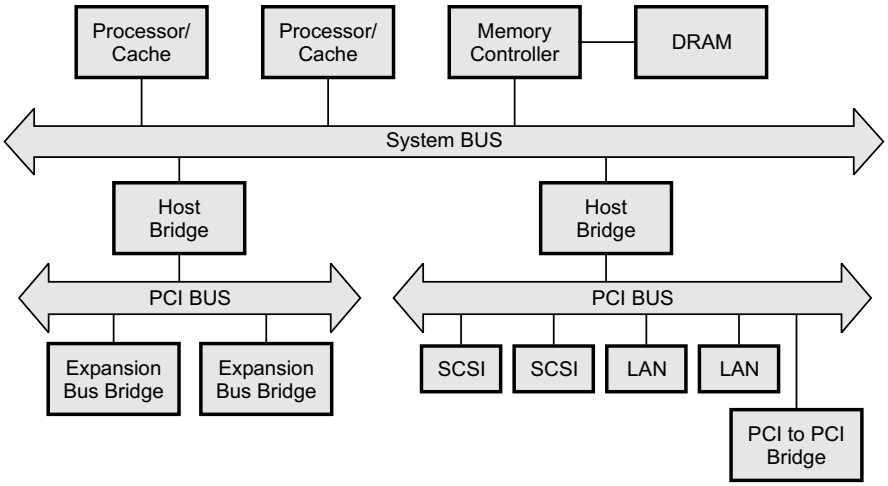
Notice that the processor bus is separate and independent of the PCI bus. The processor connects to the PCI bus through an integrated circuit called a PCI bridge. The memory controller and PCI bridge provides tight coupling with the processor and delivers data at high speeds.

Fig. 5.2.32 (b) shows a typical use of PCI in a multiprocessor system. As shown in the Fig. 5.2.32 in multiprocessor systems one or more PCI configurations may be connected by bridges to the processor's system bus. Again, the use of bridges keeps the PCI independent of the processor speed yet provides the ability to receive and deliver data rapidly.



(a) Conceptual diagram of PCI bus for single processor system

Fig. 5.2.32



(b) Conceptual diagram of PCI bus for multiprocessor system

Fig. 5.2.32

5.2.5.3 PCI Bus Signals

Fig. 5.2.34 shows pinouts for 5 V PCI bus. The PCI bus may be configured as a 32 or 64-bit bus. Fig. 5.2.33 shows 32-bit and 64-bit connectors for PCI bus.

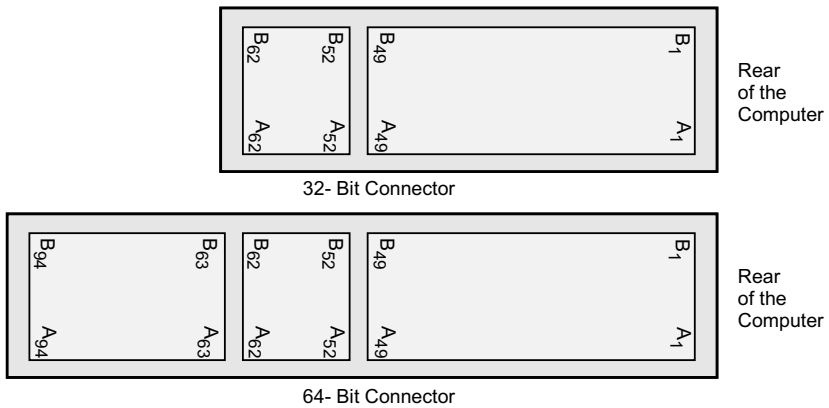


Fig. 5.2.33 The 5 V PCI slot and card configuration

PCI signals are functionally divided into the following groups :

- **System Signals :** System signals include clock and reset signals (Test clock, clock, Test Reset, Reset).
- **Address and Data Signals :** Address and data signals include 64 lines that are time - multiplexed for addresses and data lines (AD0 - AD63). The other signals from this group such as parity, command and byte enable signals are used to interpret and validate the signal lines that carry the addresses and data (PAR, C/BE).

Interface Control Signals : Interface control signals control the timing of transactions and provide co-ordination among initiators and targets. These are as follows :

$\overline{\text{FRAME}}$: The current master uses this signal to indicate the start and duration of a transaction. The signal is asserted at the start and deasserted when the initiator is ready to begin the final data phase.

$\overline{\text{IRDY}}$: The current master (initiator) uses this signal to indicate that it is ready to read or write valid data.

$\overline{\text{TRDY}}$: The target device (selected device) uses this signal to indicate that it is ready to read or write valid data.

$\overline{\text{STOP}}$: This signal indicates that the current target wishes the initiator to stop the current transaction.

$\overline{\text{LOCK}}$: The signal indicates an automatic operation that may require multiple transactions.

IDSEL : This initialization device select signal is used as a chip select during configuration read and write transactions.

$\overline{\text{DEVSEL}}$: This device select signal is used by the target to indicate device has selected.

- **Arbitration Signals :** Unlike other PCI signals, these are not shared lines. Rather, each PCI master has its own Request and Grant arbitration signals that connect it directly to the PCI bus arbiter.
- **Error Reporting Signals :** These are used to report parity and system errors ($\overline{\text{PERR}}$, $\overline{\text{SERR}}$).
- **Interrupt Signals :** Like bus arbitration signals these are not shared signals. Rather, each PCI device has its own interrupt signals ($\overline{\text{INTA}}$, $\overline{\text{INTB}}$, $\overline{\text{INTC}}$, $\overline{\text{INTD}}$).
- **Cache Support Signals :** These signals support snoopy cache protocol ($\overline{\text{SBO}}$, SDONE).
- **JTAG/Boundary Scan Signals :** These signals support testing procedures defined in IEEE standard 149.1 (TCK - Test clock, TDI - Test Input, TDO - Test Output, TMS - Test Mode Select, TRST - Test Reset).

Back of computer

Pin #	SOLDER SIDE		COMPONENT SIDE
1	–12 V	$\overline{\text{TRST}}$	
2	TCK	+12 V	
3	GND	TM5	
4	TD0	TD1	
5	+5 V	+5 V	
6	+5 V	$\overline{\text{INTA}}$	
7	$\overline{\text{INTB}}$	$\overline{\text{INTC}}$	
8	$\overline{\text{INTD}}$	+5 V	
9	$\overline{\text{PRSNT 1}}$		
10		+VI/O	
11	$\overline{\text{PRSNT 2}}$		
12	KEY	KEY	
13	KEY	KEY	
14			
15	GND	$\overline{\text{RST}}$	
16	CLK	VI/O	
17	GND	$\overline{\text{VNT}}$	
18	$\overline{\text{REQ}}$	GND	
19	+V IO		
20	AD31	AD30	
21	AD29	+ 3.3 V	
22	GND	AD28	
23	AD27	AD26	
24	AD25	GND	
25	+ 3.3 V	AD24	
26	D/$\overline{\text{BE 3}}$	IDSEL	
27	AD23	+ 3.3 V	
28	GND	AD22	
29	AD21	AD20	
30	AD19	GND	
31	+ 3.3 V	AD18	
32	AD17	AD16	
33	C/$\overline{\text{BE 2}}$	+ 3.3 V	
34	GND	$\overline{\text{FRAME}}$	
35	$\overline{\text{IRDY}}$	GND	
36	+ 3.3 V	$\overline{\text{TRDY}}$	
37	$\overline{\text{DEVSEL}}$	GND	
38	GND	$\overline{\text{STOP}}$	
39	$\overline{\text{LOCK}}$	+ 3.3 V	
40	$\overline{\text{PERR}}$	SDONE	

Pin #	SOLDER SIDE		COMPONENT SIDE
41	+3.3 V	$\overline{\text{SBO}}$	
42	$\overline{\text{SERR}}$	GND	
43	+3.3 V	PAR	
44	C/$\overline{\text{BE1}}$	AD15	
45	AD14	+3.3 V	
46	GND	AD13	
47	AD12	AD11	
48	AD10	GND	
49	GND	AD9	
50	KEY	KEY	
51	KEY	KEY	
52	AD8	C/$\overline{\text{BE0}}$	
53	AD7	+3.3 V	
54	+3.3 V	AD6	
55	AD5	AD4	
56	AD3	GND	
57	GND	AD2	
58	AD1	AD0	
59	+V IO	+V IO	
60	$\overline{\text{ACK64F}}$	$\overline{\text{REQ64}}$	
61	+5 V	+5 V	
62	+5 V	+5 V	
63		GND	
64	GND	C/$\overline{\text{BE 7}}$	
65	C/$\overline{\text{BE 6}}$	C/$\overline{\text{BE 5}}$	
66	C/$\overline{\text{BE 4}}$	+V IO	
67	GND	PAR64	
68	AD63	AD 62	
69	AD 61	GND	
70	+V IO	AD60	
71	AD59	AD58	
72	AD57	GND	
73	GND	AD56	
74	AD55	AD54	
75	AD53	+V IO	
76	GND	AD52	
77	AD51	AD50	
78	AD49	GND	
79	+V IO	AD48	
80	AD47	AD46	
81	AD45	GND	
82	GND	AD44	
83	AD43	AD42	
84	AD41	+V I/O	
85	GND	AD40	
86	AD39	AD38	
87	AD37	GND	
88	+V I/O	AD36	
89	AD35	AD34	
90	AD33	GND	
91	GND	AD32	
92			
93		GND	
94	GND		

Note : (1) Pins 63-94 exist only on the 64 bit PCI card.
(2) +V I/O is 3.3 V on a 3.3 V board and +5 V on a 5 V board.
(3) Blank pins are reserved.

Fig. 5.2.34 The pin-out of the PCI bus

5.2.5.4 PCI Bus Commands

The PCI bus commands are used to carryout transactions between an initiator, or master, and a target. These commands are :

- **INTA Sequence :** INTA sequence is a read command intended for the interrupt controller on the PCI bus. The byte - sized interrupt vector is returned during a byte read operation.
- **Special Cycle :** This command is used by the initiator to send a message to one or more targets.
- **I/O Read Cycle :** I/O read cycle command is used to transfer data from an I/O device to the initiator.
- **I/O Write Cycle :** I/O write cycle command is used to transfer data from an initiator to the I/O device.
- **Memory Read Cycle :** Memory read cycle command is used to transfer data from an memory device on the PCI bus to the initiator.
- **Memory Write Cycle :** Memory write cycle command is used to transfer data from an initiator to the memory device on the PCI bus.
- **Configuration Read :** This command is used to read configuration information from the PCI device.
- **Configuration Write :** This command is used to write configuration information in the PCI device.
- **Memory Multiple Access :** This is similar to the memory read cycle command, except that it is usually used to access many data instead of one.
- **Dual Addressing Cycle :** This command is used for transferring address information to a 64-bit PCI device, which only contains a 32 - bit data path.
- **Line Memory Access :** This command is used to read more than two 32 - bit numbers from the PCI bus.
- **Memory Write with Invalidation :** This command is used to transfer data in one or more cycles to memory.

5.2.5.5 Data Transfer

Every data transfer on the PCI bus is a single transaction consisting of one address phase and one or more data phases. Let us see the typical read cycle on the PCI bus. All the events during read cycle are synchronized with the falling edge of the clock cycle. The devices connected to the bus sample the bus lines on the rising edge at the beginning of a bus cycle. The events during the bus cycle are explained below and they are also labeled on the diagram.

1. The master asserts $\overline{\text{FRAME}}$ signal to indicate the start of a transaction. This signal remains asserted until initiator is ready to begin the data phase. The initiator puts the start address on the address bus, and initiates read cycle by activating C/$\overline{\text{BE}}$ lines.

2. At the beginning of clock 2, the target device recognizes its address on the address bus.
3. The initiator ceases the address lines and it changes the information on the C/$\overline{BE}$ lines to designate which address lines are to be used for transfer for the currently addressed data from 1 to 4 bytes. The initiator also asserts IRDY to indicate that it is ready for the first data byte.
4. The selected target asserts $\overline{DEVSEL}$ to indicate that it has recognized its address and it responds by placing the requested data on the address bus. It then asserts $\overline{TRDY}$ to indicate that valid data is present on the bus.
5. The initiator reads the data at the beginning of clock 4 and changes the byte enable lines as needed in preparation for the next read. If target needs some more time to put next byte of data on the address lines, it informs this by deasserting $\overline{TRDY}$ line. Accordingly, the initiator does not read the data lines at the beginning of the next clock cycle and does not change byte enable during that cycle. This is illustrated in the fifth clock cycle.
6. It may happen that, the target places the data byte but the initiator is not ready to accept that data byte. The initiator indicates this by deasserting $\overline{IRDY}$ signal.
7. The initiator deasserts $\overline{FRAME}$ signal indicating that it is last data byte transfer.

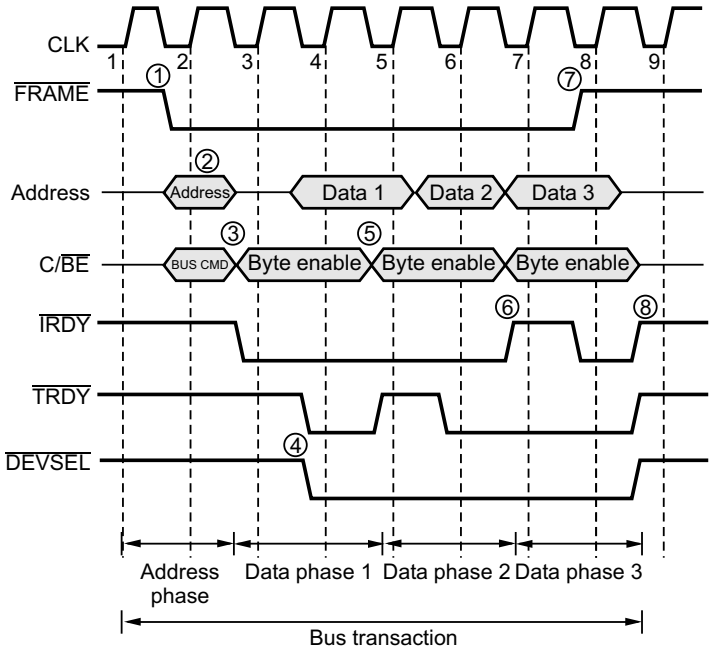


Fig. 5.2.35 Timing diagram for PCI read operation

8. The initiator deasserts $\overline{IRDY}$ and the target deasserts $\overline{TRDY}$ and $\overline{DEVSEL}$, returning bus to the idle state.

Similar kind of handshaking signals are activated during PCI write operation. Here, the data from initiator is copied into the targeted devices.

5.2.5.6 PCI Arbitration

As mentioned earlier PCI bus uses centralized, synchronous arbitration in which each master has a unique request (REQ) and grant (GNT) signals. These signals are connected to the central bus arbiter, as shown in the Fig. 5.2.36.

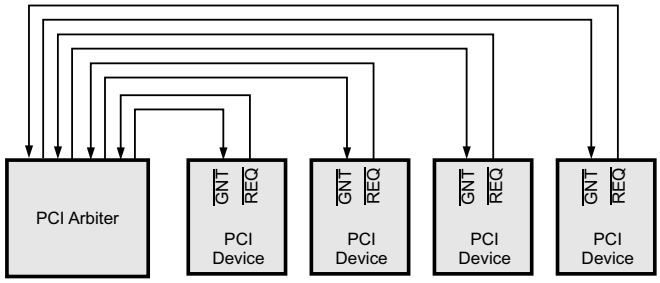


Fig. 5.2.36 PCI bus arbiter

The PCI bus does not specify a particular arbitration algorithm. Therefore, it allows to use a first in first serve approach, a round-robin approach or priority approach. Let us see the operation of PCI arbiter with two bus masters in the system.

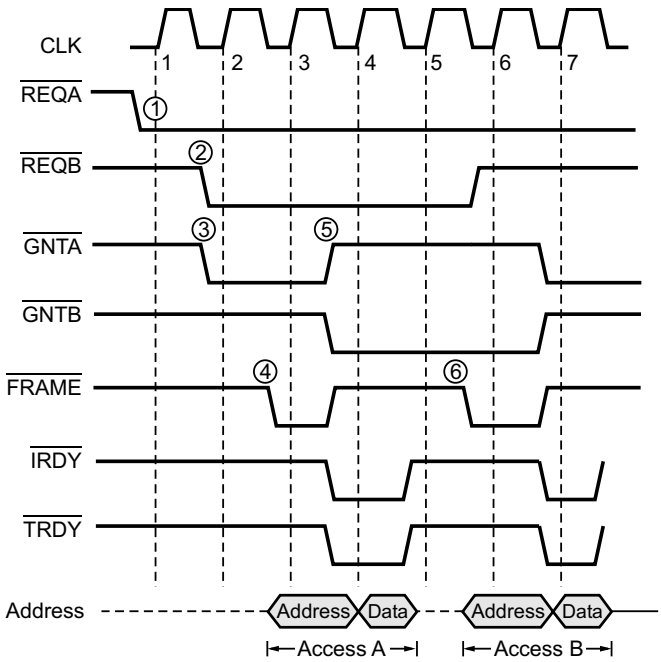


Fig. 5.2.37 PCI bus arbitration between two masters

1. Let us assume that Master A has asserted its $\overline{REQ}$ ($\overline{REQA}$) signal before the start of clock 1. The arbiter samples that signal at the beginning of clock 1.

2. The Master B request for bus by asserting its $\overline{REQ}$ ($\overline{REQB}$) signal during clock 1.
3. At the same time, the arbiter asserts $\overline{GNTA}$ signal to grant bus access to Master A.
4. The Master A samples $\overline{GNTA}$ signal at the beginning of clock 2 and recognizes that it has been granted bus access. At this time $\overline{IRDY}$ and $\overline{TRDY}$ are deasserted, indicating that the bus is idle. Therefore, Master A asserts $\overline{FRAME}$ and places the address information on the address bus and the command on the C/$\overline{BE}$ bus (it is not shown in the Fig. 5.2.37). The MASTER A continues to assert $\overline{REQA}$ signal if it has to perform another transaction as shown in the timing diagram.
5. At the beginning of clock 3, the bus arbiter samples all $\overline{GNT}$ lines and makes an arbitration decision according to arbitration algorithm used. In this timing diagram it grants the bus for master B for the next transaction. It then deasserts $\overline{GNTA}$, $\overline{IRDY}$, $\overline{TRDY}$ and $\overline{FRAME}$ signals and asserts $\overline{GNTB}$. However, Master B will not be able to use the bus until it returns to an idle state.
6. At the beginning of clock 5, Master B finds $\overline{IRDY}$ and $\overline{FRAME}$ deasserted and so it is allowed to take control of the bus by asserting $\overline{FRAME}$. It also deasserts the $\overline{REQB}$ line because Master B has to perform only one transaction.

Review Questions

1. *How does bus arbitration typically work ?* **May-06, Marks 2**
2. *Discuss the data transfer mechanism of the PCI bus.* **May-06, Marks 6**
3. *Write short note on PCI.* **Dec.-06, Marks 8**
4. *Write short note on bus arbitration.* **Dec.-07, Marks 4**
5. *What factors influences the bus design decisions ?* **May-08, Marks 2**
6. *Define bus arbitration.* **May-08, Marks 2**
7. *Explain handshake protocol. Depict clearly how it controls data transfer during an input operation.* **May-08, Marks 2**
8. *With a neat diagram describe the implement of distributed arbitration.* **May-09, Marks 6**
9. *What are handshaking signals and what are the sequences of events during an input operation using handshake scheme ?* **May-09, Marks 8**
10. *Define control Bus.* **May-10, Marks 2**
11. *Explain various bus arbitration mechanisms in DMA transfer.* **May-10, Marks 16**
12. *Discuss the data transfer mechanism of the PCI bus.* **May-10, Marks 16**
13. *What is mean by bus arbitration ? Describe bus arbitration daisy chaining and polling schemes for bus arbitration in detail.* **May-11, Marks 10**
14. *Explain different types of bus arbitration scheme.* **Dec.-11, 12, Marks 8**

5.3 I/O and System Control

May-06, 07,08, 09, 10, 12, Dec.-06, 07, 08, 10, 11, 12

The important components of any computer system are, CPU, memory, and I/O devices (peripherals). The CPU fetches instructions (opcodes and operands/data) from memory, processes them and stores results in memory. The other components of the computer system (I/O devices) may be loosely called the Input/Output system. The main function of I/O system is to transfer information between CPU or memory and the outside world.

The important point to be noted here is, I/O devices (peripherals) cannot be connected directly to the system bus. The reasons are discussed here.

1. A variety of peripherals with different methods of operation are available. So it would be impractical to incorporate the necessary logic within the CPU to control a range of devices.
2. The data transfer rate of peripherals is often much slower than that of the memory or CPU. So it is impractical to use the high speed system bus to communicate directly with the peripherals.
3. Generally, the peripherals used in a computer system have different data formats and word lengths than that of CPU used in it.

So to overcome all these difficulties, it is necessary to use a module in between system bus and peripherals, called I/O module or I/O system, or I/O interface.

5.3.1 I/O Modules

Let us see the major functions and the structure of I/O module.

I/O module has two major functions :

1. Interface to the CPU and memory via the system bus,
2. Interface to one or more I/O devices by tailored data links.

For clear understanding refer Fig. 5.3.1.

Links to peripheral devices are used to exchange control, status and data between the I/O module and the external devices. An important point that is to be noted here is, an I/O module is not simply mechanical connectors for connecting different devices required in the system to the system bus. It contains some logic for performing function of communication between the peripheral (I/O device) and the bus. The I/O module must have an interface internal to the computer (to the CPU and memory) and an interface external to the computer (to the external I/O devices).

5.3.1.1 Major Requirements of an I/O Module

The major requirements for an I/O module can be given as

1. Control and timing
2. CPU communication
3. Device communication
4. Data buffering
5. Error detection

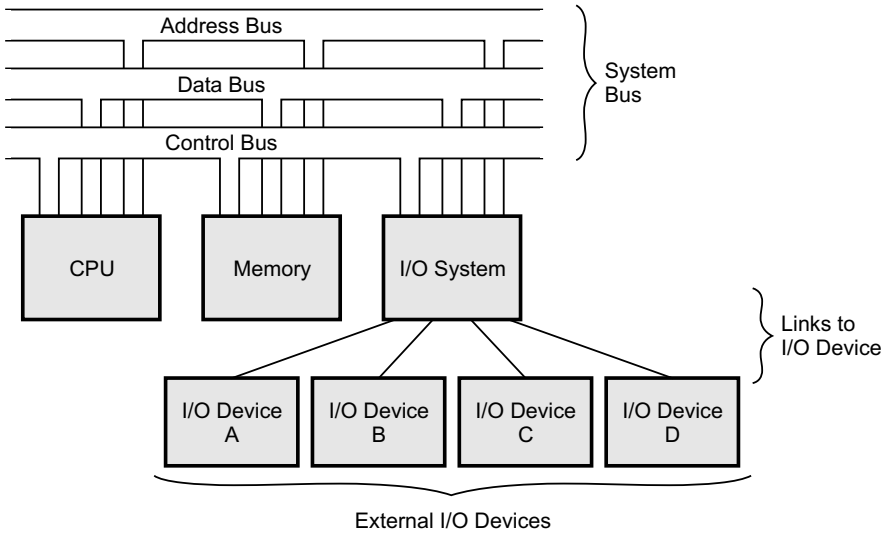


Fig. 5.3.1 Generic model of computer system

All these requirements are explained here. The I/O module includes a control and timing requirements to co-ordinate the flow of traffic between internal resources (memory, system bus) and external devices. CPU communication involves different types of signal transfers such as

- CPU sends commands to the I/O module which are generally the control signals on the control bus.
- Exchange of data between the CPU and the I/O module over the data bus.
- The data transfer rate of peripherals is often much slower than that of the CPU. So it is necessary to check whether peripheral is ready or not for data transfer. If not, CPU must wait. So it is important to know the status of I/O module. The status signals such as BUSY, READY can be used for this purpose.
- A number of peripheral devices may be connected to the I/O module. The I/O module controls the communication of each peripheral with CPU. So it must recognize one unique address for each peripheral connected to it.

The I/O module must able to perform device communication which involves commands, status information and data.

Data buffering is also an essential task of an I/O module. Data transfer rates of peripheral devices are quite high than that of CPU and memory. The data coming from memory or CPU are sent to an I/O module, buffered and then sent to the peripheral device at its data rate. Also, data are buffered in I/O module so as not to tie up the memory in a slow transfer operation. Thus the I/O module must be able to operate at both peripheral and memory speeds.

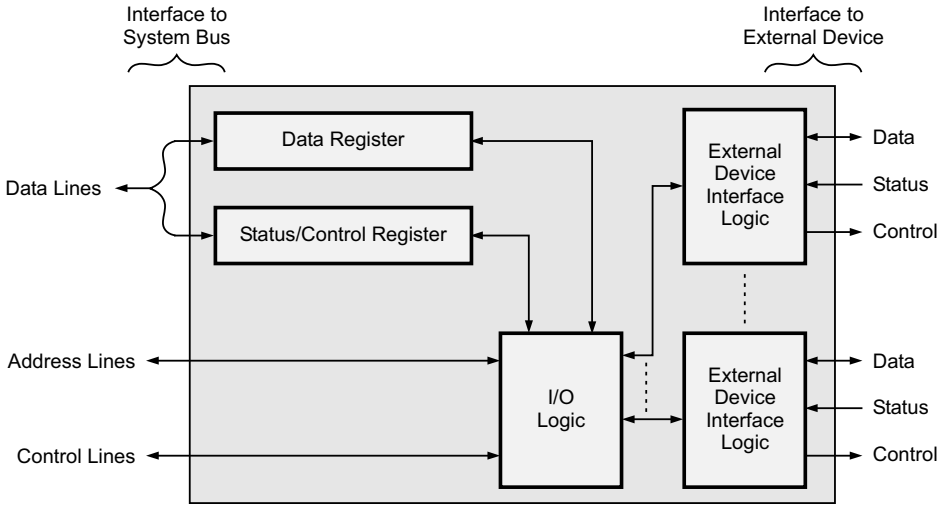


Fig. 5.3.2 Block diagram of an I/O module

I/O module is also responsible for error detection and for reporting errors to the CPU. The different types of errors are mechanical, electrical malfunctions reported by the device such as bad disk track, unintentional changes to the bit pattern, transmission errors etc. To fulfil all these requirements the important blocks necessary in any I/O module are shown in Fig. 5.3.2.

In a computer system, if there is a high-level interface of I/O module to the CPU, and if that I/O module takes on most of the detailed processing burden, then it is usually referred to as an I/O channel or I/O processor. Generally I/O channels are used on mainframes, with minicomputers employing a mixture. An I/O module that is quite primitive and requires detailed control is usually referred to as an I/O controller or device controller. I/O controllers are generally used on microcomputers.

5.3.2 Programmed I/O

I/O operations will mean a data transfer between an I/O device and memory or between an I/O device and the CPU. If in any computer system I/O operations are completely controlled by the CPU, then that system is said to be using '**programmed I/O**'. When such a technique is used, CPU executes programs that initiate, direct and terminate the I/O operations, including sensing device status, sending a read or write command and transferring the data. It is the responsibility of the CPU to periodically check the status of

the I/O system until it finds that the operation is complete. Let us consider the following example.

The processor's software checks each of the I/O devices every so often. During this check, the microprocessor tests to see if any device needs servicing. Fig. 5.3.3 shows the flow chart for this. This is a simple program which services I/O ports A, B and C. The routine checks the status of I/O ports in proper sequence. It first transfers the status of I/O port A into the accumulator. Then the routine block checks the contents of accumulator to see if the service request bit is set. If it is, I/O port A service routine is

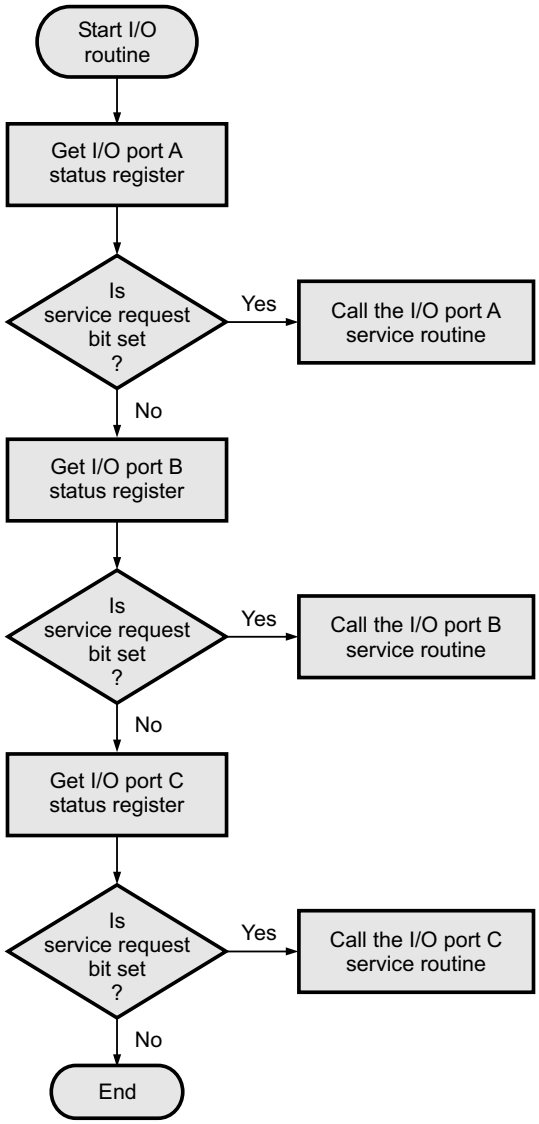


Fig. 5.3.3 Flowchart for I/O service routine

called. After completion of service routine for I/O port A, the polling routine moves on to test port B and the process is repeated. This test and service procedure continues until all the I/O port status registers are tested and all the I/O ports requesting service are serviced. Once this is done, the processor continues to execute the normal programs.

The routine assigns priorities to the different I/O devices. Once the routine is started, the service request bit at port A is always checked first. Once port A is checked, port B is checked, and then port C. However, the order can be changed by simply changing the routine and thus the priorities.

In systems with programmed I/O I/O devices, memory and CPU normally communicate via a common shared bus called the system bus. The address lines of that bus are used to select memory locations and I/O devices. A number of I/O devices can be used in a system. For each device, address is different. How these devices are accessed is an important point which is discussed in further section.

5.3.2.1 I/O Addressing

The most of the processors support isolated I/O system. It partitions memory from I/O, via software, by having instructions that specifically access (address) memory and others that specifically access I/O. When these instructions are decoded by the processor, an appropriate control signal is generated to activate either memory or I/O operation. I/O devices can be interfaced to a computer system I/O in two ways, which are called interfacing techniques,

- Memory mapped I/O
- I/O mapped I/O

Memory mapped I/O

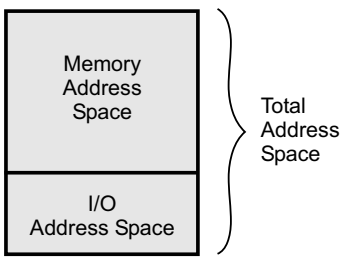


Fig. 5.3.4 Address space

In this technique, the total memory address space is partitioned and part of this space is devoted to I/O addressing as shown in Fig. 5.3.4.

When this technique is used, a memory reference instruction that causes data to be fetched from or stored at address specified, automatically becomes an I/O instruction if that address is made the address of an I/O port. The usual memory related instructions are used for I/O related operations. The special I/O instructions are not required. Fig. 5.3.5 gives the programmed I/O with memory mapped I/O.

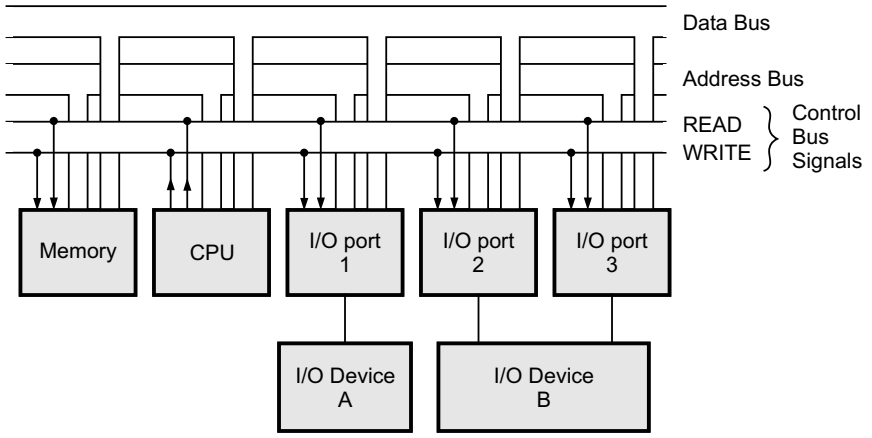


Fig. 5.3.5 Programmed I/O with memory mapped I/O

I/O mapped I/O

If we do not want to reduce the memory address space, we allot a different I/O address space, apart from total memory space which is called I/O mapped I/O technique as shown in Fig. 5.3.6.

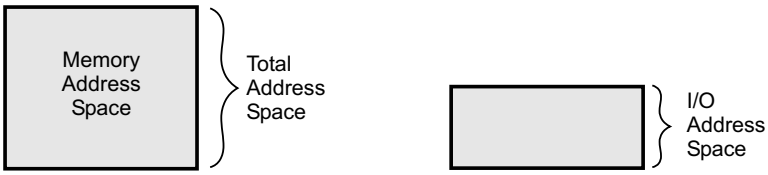


Fig. 5.3.6 Address space

Here, the advantage is that the full memory address space is available. But now, the memory related instructions do not work. Therefore, processor can only use this mode if it has special instructions for I/O related operations such as I/O read, I/O write.

Fig. 5.3.7 gives I/O with I/O mapped I/O technique.

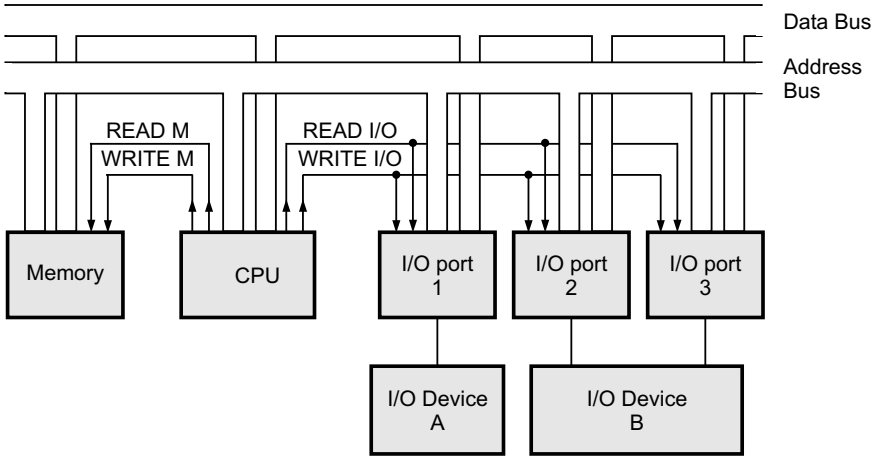


Fig. 5.3.7 Programmed I/O with I/O mapped I/O

Memory mapped I/O, I/O mapped I/O comparison

Sr. No.	Memory mapped I/O	I/O mapped I/O
1.	Memory and I/O share the entire address range of processor.	Processor provides separate address range for memory and I/O devices.
2.	Usually, processor provides more address lines for accessing memory. Therefore more decoding is required control signals.	Usually, processor provides less address lines for accessing I/O. Therefore, less decoding is required.
3.	Memory control signals are used to control read and write I/O operations.	I/O control signals are used to control read and write I/O operations.

5.3.2.2 I/O Instructions

When programmed I/O technique is used, CPU fetches I/O related instructions from memory and issues I/O commands to I/O system to execute the instruction. They form of the instruction depends on the technique used for I/O addressing, i.e. memory mapped I/O or I/O mapped I/O.

As seen before, using memory mapped I/O technique, a memory reference instruction that causes data to be fetched from or stored at address specified automatically becomes an I/O instruction if that address is devoted to an I/O port. The usual memory load and store instructions are used to transfer a word of data to or from an I/O port.

When I/O mapped I/O technique is used, separate I/O instructions are required to activate READ I/O and WRITE I/O lines, which cause a word to be transferred between the addressed I/O port and the CPU. The processor has two separate instructions, IN and OUT for I/O data transfer, e.g. the Intel 8085 microprocessor has two main I/O instructions. The *IN port address* instruction causes a word to be transferred from I/O port having address specified within the instruction to register A (accumulator) of 8085. The instruction *OUT port address* transfers a word from register A to I/O port having address specified within the instruction.

When an I/O instruction is encountered by the CPU, the addressed I/O port is expected to be ready to respond the instruction to prevent loss of information. Thus it is desirable for the CPU to know the I/O device status (ready or not for I/O data transfer). In programmed I/O systems, the CPU is usually programmed to test the I/O device status before initiating a data transfer.

5.3.2.3 Interfacing

As shown in Fig. 5.3.8 , I/O devices can be interfaced to CPU using circuits such as latches and buffers.

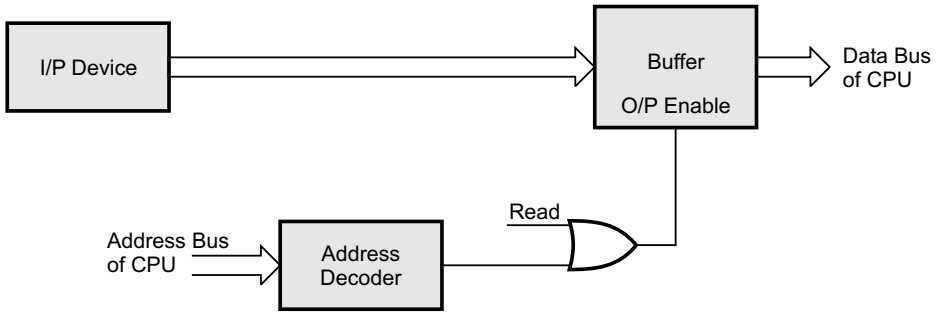
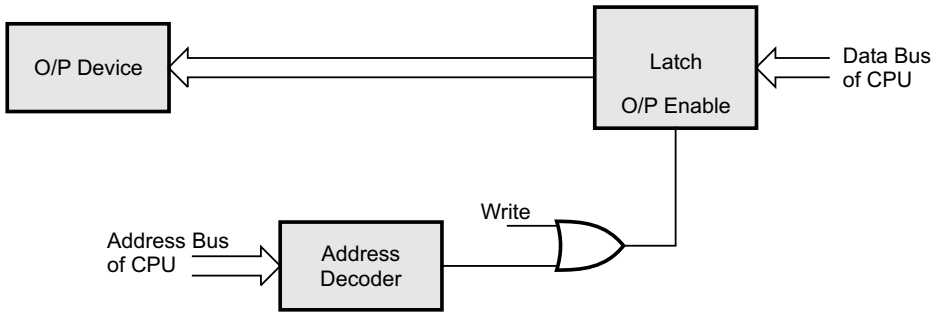


Fig. 5.3.8 (a) Input interfacing

Typically, there will be number of I/O devices connected to the system. Each device is given an unique address. When the CPU issues an I/O command, the command contains the address of the desired device. According to the address placed on the address bus, the device is selected. The CPU either reads data through buffer from input device addressed or writes data through latch to output device addressed. However, latches and buffers once hardwired, perform only one function (either as input port if it is a buffer and as output port if it is a latch). Thus limiting their capabilities.



(b) Output interfacing

Fig. 5.3.8 I/O interfacing

To improve the performance of I/O interfaces some manufacturers have designed programmable I/O interfaces. Along with the I/O functions these integrated circuits (ICs) or chips can perform many other functions such as time delays, counting, interrupt etc. This IC is a combination of many devices on a single chip. The biggest advantage of it is that it can be set up to perform specific functions by writing a code in its internal register. For clear understanding of I/O interfacing device, let us consider a programmable peripheral interface 8255A.

Programmable peripheral interface 8255A

Features of 8255A

1. The 8255A is a widely used, programmable, parallel I/O interface.
2. It can be programmed to transfer data under various conditions, from simple I/O to interrupt I/O.
3. It has three 8-bit ports : Port A, Port B and Port C, which are arranged in two groups of 12 pins. Each port has an unique address, and data can be read from or written to a port. The unique address is also assigned to the control register into which control words are written for programming the 8255 to operate in various modes.
4. The 8255 can operate in 3 modes : i) Mode 0, ii) Mode 1, and iii) Mode 2.

Fig. 5.3.9 shows the block diagram of 8255A. It mainly consists of data bus buffer, control logic and group A and group B controls.

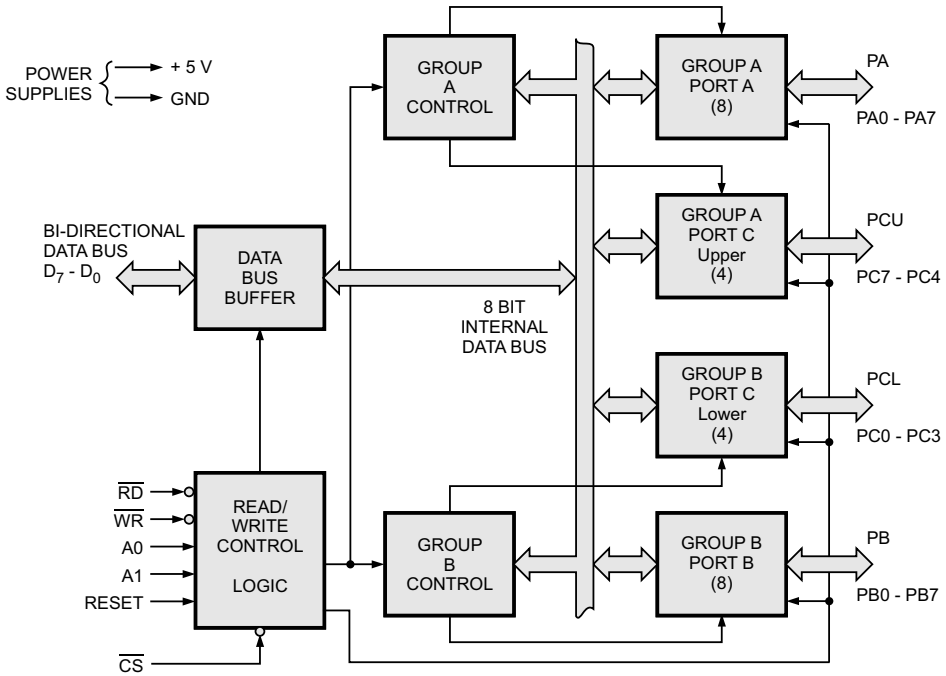


Fig. 5.3.9 Block diagram of 8255A

The input/output instructions executed by the CPU either read data from or write data into the tri-state bidirectional buffer. The read/write control logic block accepts control signals as well as inputs from the address bus and issues commands to the individual group control blocks (Group A control and group B control). A_0 and A_1 signals are used for the selection of one of the three ports or control/status register.

Each of the group A and group B control blocks receives control words from the CPU and issues appropriate commands to the ports associated with it.

Port A has an 8-bit latched and buffered output and an 8-bit input latch. Port B has an 8-bit data I/O latch/buffer and an 8-bit data input buffer. Port C has one 8-bit unlatched input buffer and an 8-bit output latch/buffer. Port C can be splitted into two parts of 4 bits as Port C_{upper} and Port C_{lower}

Operating modes

The 8255 can operate in three modes :

In Mode 0, Port A and Port B can be configured as simple 8-bit input or output ports without handshaking. The two halves of Port C can be programmed separately as 4-bit input or output ports.

In Mode 1, two groups each of 12 pins are formed. Group A consists of Port A and the upper half of Port C while Group B consists of Port B and the lower half of Port C. Ports A and B can be programmed as 8-bit Input or Output ports with three lines of Port C in each group used for handshaking.

In Mode 2, only Port A can be used as a bi-directional port. The handshaking signals provided on five lines of Port C (PC_3 – PC_7). Port B can be used in Mode 0 or in Mode 1.

The speed of operation or data transfer rate of peripheral devices is much slower than that of CPU or memory. In programmed I/O, CPU has to wait a long time for the I/O system of concern to be ready for data transfer (reception or transmission). As a result, the performance of the entire system is severely degraded. The alternative for programmed I/O is use of interrupts.

5.3.3 Interrupt Driven I/O

Sometimes it is necessary to have the computer automatically execute one of a collection of special routines whenever certain conditions exists within a program or the computer system e.g. It is necessary that computer system should give response to devices such as keyboard, sensor and other components when they request for service.

The most common method of servicing such device is the **polled approach**. This is where the processor must test each device in sequence and in effect "ask" each one if it needs communication with the processor. It is easy to see that a large portion of the main program is looping through this continuous polling cycle. Such a method would have a serious and decremental effect on system throughput, thus limiting the tasks that could be assumed by the computer and reducing the cost effectiveness of using such devices.

A more desirable method would be the one that allows the processor to execute its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method, would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is completed, the processor would resume exactly where it left off. This method is called **interrupt method.** It is easy to see that system throughput would drastically increase, and thus enhance its cost effectiveness. Most processors allow execution of special routines by interrupting normal program execution. When a processor is interrupted, it stops executing its current program and calls a special routine which "services" the interrupt. This is illustrated in Fig. 5.3.10. The event that causes the interruption is called **interrupt** and the special routine executed to service the interrupt is called **interrupt service routine/procedure**. Normal program can be interrupted by three ways :

1. By external signal
2. By a special instruction in the program or
3. By the occurrence of some condition.

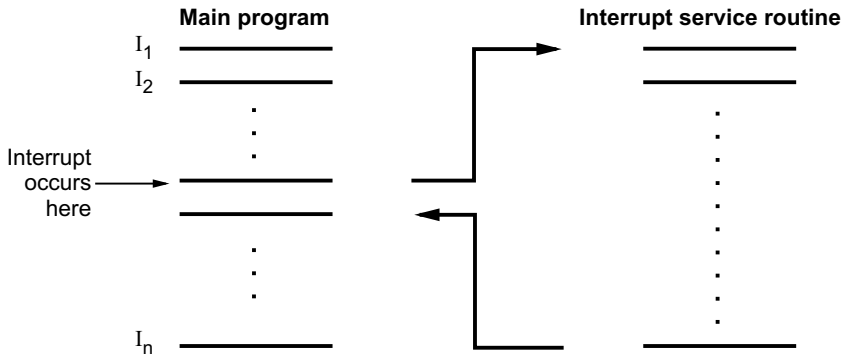


Fig. 5.3.10 Interrupt operation

An interrupt caused by an external signal is referred as a **hardware interrupt**. Conditional interrupts or interrupts caused by special instructions are called **software interrupts**.

5.3.3.1 Interrupt Hardware

An I/O device requests an interrupt by activating a bus line called **interrupt request** or simply **interrupt**. The interrupts are classified as :

1. Single level interrupts
2. Multi level interrupts

Single Level Interrupts

In a single level interrupts there can be many interrupting devices. But all interrupt requests are made via a single input pin of the CPU. When interrupted, CPU has to poll the I/O ports to identify the request device. Polling software routine that checks the logic state of each device. Once the interrupting I/O port is identified, the CPU will service it and then return to task it was performing before the interrupt.

Fig. 5.3.11 shows single level interrupt system, in which the interrupt requests from all the devices are logically ORed and connected to the interrupt input of the processor. Thus, the interrupt request from any device is routed to the processor interrupt input. After getting interrupted, processor identifies requesting device by reading interrupt status of each device.

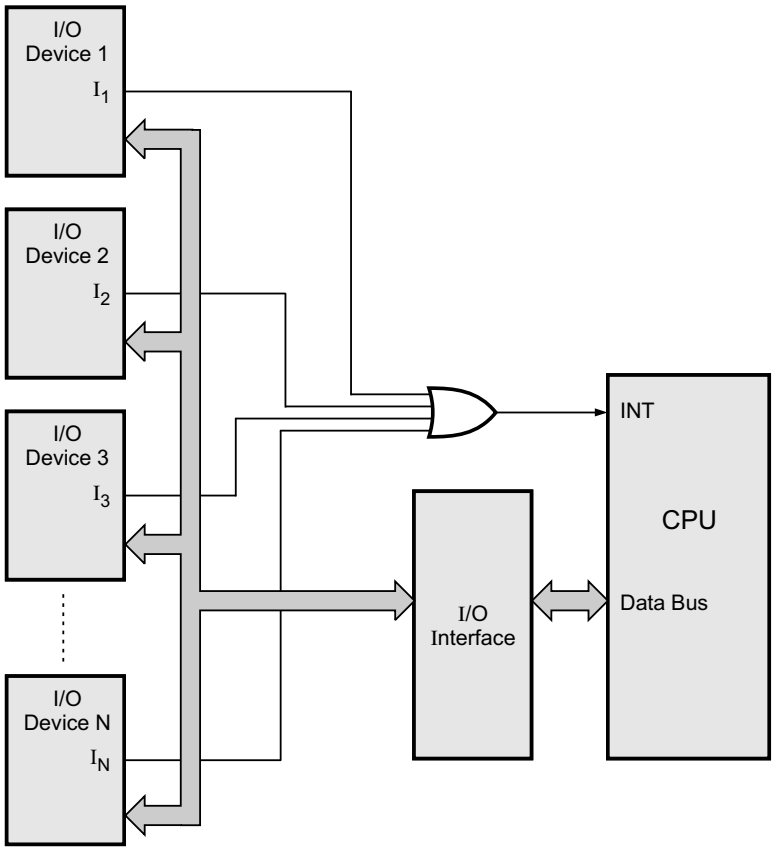


Fig. 5.3.11 Single level interrupt system

The Fig. 5.3.12 shows the equivalent circuit for single interrupt-request line used to serve n-devices. This is another way of representing single level interrupt system. Here, all devices are connected to the $\overline{INTR}$ line via switches to ground. To request an interrupt, a device closes its associated switch. When all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt request line will be equal to V_{DD}. This is the inactive state of the line. When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt-request signal, INTR, received by the processor to go to 1. Closing of one or more switching will cause the line voltage to drop to 0, resulting INTR = 1. This is equivalent to logically OR of the requests from individual devices to generate INTR signal.

To implement electronic switch shown in the Fig. 5.3.12, we have to use open collector or open drain gates. Because output of an open collector or an open drain gate is equivalent to a switch to ground that is open, and outputs from more than one gate can be connected together and tied to V_{CC} or V_{DD}.

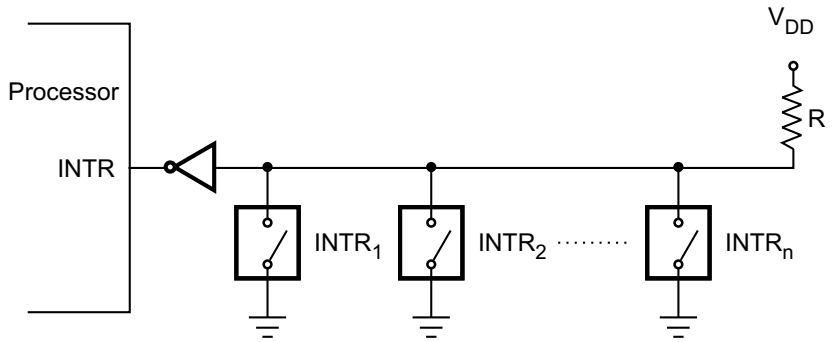


Fig. 5.3.12

Multi Level Interrupts

In multi level interrupts, processor has more than one interrupt pins. The I/O devices are tied to the individual interrupt pins. Thus, the interrupts can be immediately identified by the CPU upon receiving an interrupt request from it. This allows processor to go directly to that I/O device and service it without having to poll first. This obviously saves the time in processing interrupt. Fig. 5.48 shows the multi level interrupt system.

In multi level interrupt system, when a processor is interrupted, it stops executing its current program and calls a special routine which "services" the interrupt. The event that causes the interruption is called **interrupt** and the special routine which is executed is called **interrupt service routine**. When the external asynchronous input (interrupt input) is asserted (a signal is sent to the interrupt input), a special sequence in the control logic begins.

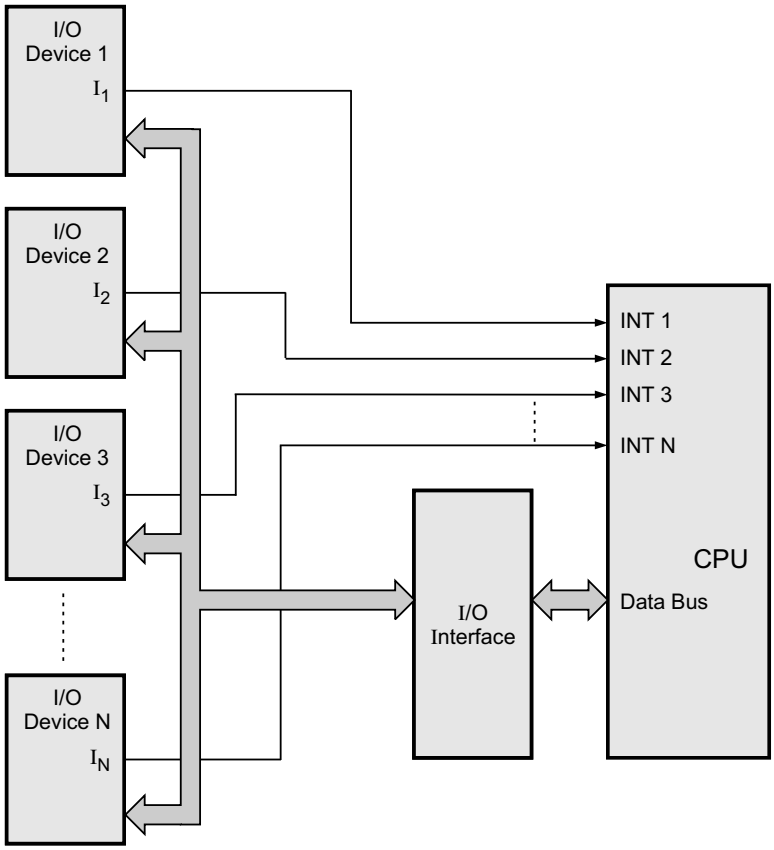


Fig. 5.3.13 Multi level interrupt system

1. The processor completes its current instruction. No instruction is cut-off in the middle of its execution.
2. The program counter's current contents are stored on the stack. Remember, during the execution of an instruction the program counter is pointing to the memory location for the next instruction.
3. The program counter is loaded with the address of an interrupt service routine.
4. Program execution continues with the instruction taken from the memory location pointed by the new program counter contents.
5. The interrupt program continues to execute until a return instruction is executed.
6. After execution of the RET instruction processor gets the old address (the address of the next instruction from where the interrupt service routine was called.) of the program counter form the stack and puts it back into the program counter. This allows the interrupted program to continue executing at the instruction following

the one where it was interrupted. Fig. 5.3.14 shows the response to an interrupt with the flowchart and diagram.

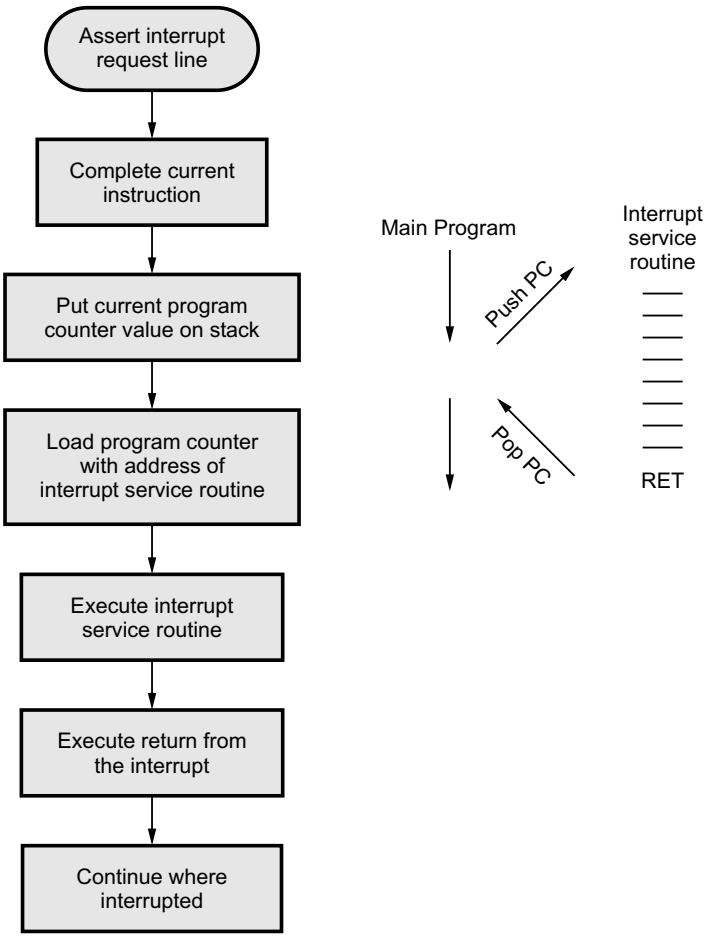


Fig. 5.3.14 Response to an interrupt with the flowchart and diagram

5.3.3.2 Enabling and Disabling Interrupts

Maskable interrupts are enabled and disabled under program control. By setting or resetting particular flip-flops in the processor, interrupts can be masked or unmasked, respectively. When masked, processor does not respond to the interrupt even though the interrupt is activated. Most of the processors provide the masking facility. In the processor those interrupts which can be masked under software control are called **maskable interrupts**. The interrupts which can not be masked under software control are called **non-maskable interrupts**.

5.3.3.3 Handling Multiple Devices

In section 5.3.3.1 we have seen hardware required for handling multiple interrupts. In this section we discuss the issues related to processing of these multiple interrupts from multiple devicing. To handle interrupts from multiple devices processor has to perform following tasks.

- It has to recognize the device requesting an interrupt.
- It has to obtain the starting address of interrupt service routine corresponds to interrupt request. Because each interrupt request has its own interrupt service routine.
- It has to allow the device to interrupt while another interrupt is being serviced.
- It has to take decision that which interrupt should be serviced first when there are simultaneous interrupt requests from two different devices.

The modern processors are capable of performing above mentioned tasks. In these processors, such capabilities are implemented by using vectoring of interrupts, by nesting of interrupts and by assigning priorities to the interrupts.

Example 5.3.1 *Design a parallel priority interrupt hardware for a system with eight interrupt sources.* **May-07, Marks 8**

Solution :

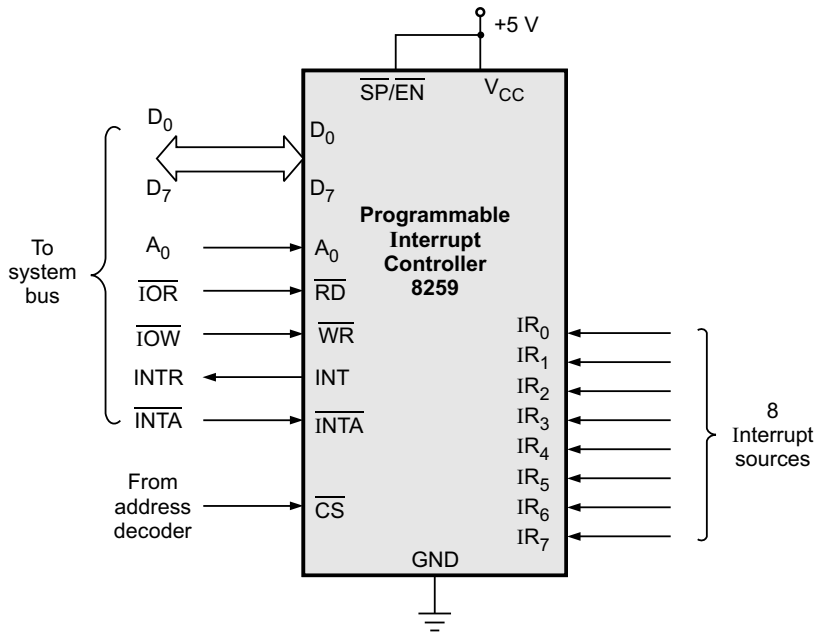


Fig. 5.3.15

5.3.3.4 Vectored Interrupts

When the external device interrupts the CPU (interrupt request), CPU has to execute interrupt service routine for servicing that interrupt. If the internal control circuit of the processor produces a CALL to a predetermined memory location which is the starting address of interrupt service routine, then that address is called **vector address** and such interrupts are called **vector interrupts**. For vector interrupts fastest and most flexible response is obtained since such an interrupt causes a direct hardware-implemented transition to the correct interrupt-handling program. This technique is called **vectoring**. Fig. 5.3.16 shows a widely used vectored interrupt scheme.

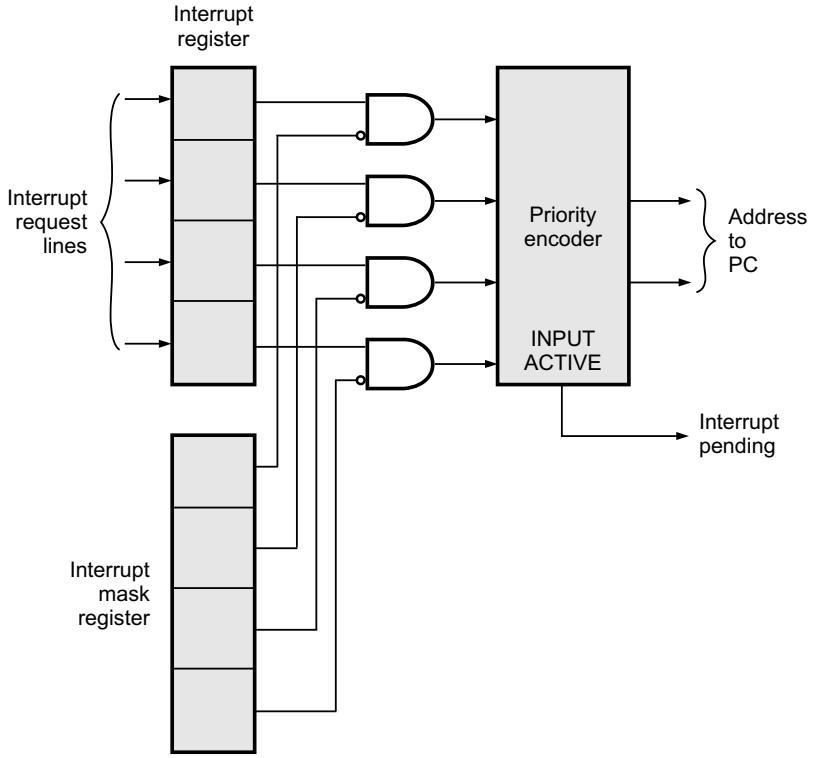


Fig. 5.3.16 A vectored interrupt scheme

As shown in Fig. 5.3.16, interrupt request signals are stored in the interrupt register. The programmable interrupt mask register is used to disable any or all of the interrupt request lines. For any AND gate, one input is the interrupt request signal from interrupt register and second input is the inverted mask condition for the same interrupt from interrupt mask register. When any interrupt is masked (1), the input to AND gate is zero, hence output goes zero. This means when interrupt is masked, the corresponding interrupt request line is disabled. When interrupt request is present for unmasked interrupt, that is fed into a priority encoder which produces the address which is then inserted into the program counter.

5.3.3.5 Nested Interrupts

For some devices, a long delay in responding to an interrupt request may cause error in the operation of computer. Such interrupts are acknowledged and serviced eventhough processor is executing an interrupt service routine for another device. A system of interrupts that allows an interrupt service routine to be interrupted is known as **nested interrupts**. Consider, for example, a computer that keeps a track of the time of day using real time clock. This real time clock requests to the processor at regular intervals and processor accordingly updates the counts for seconds, minutes and hours of the day. For the proper operation such an interrupt request from real time clock must be processed eventhough computer is executing an interrupt service for another device.

5.3.3.6 Interrupt Priority

When interrupt requests arrive from two or more devices simultaneously, the processor has to decide which request should be serviced first and which one should be delayed. The processor takes the decision with the help of interrupt priorities. It accepts the request having the highest priority.

In case polling to identify the interrupting device, priority is automatically assigned by the order in which devices are polled. Therefore, no further arrangement is required to accommodate simultaneous interrupt requests. However, in case of vectored interrupts, the priority of any device is usually determined by the way the device is connected to the processor. Most common way to connect the devices is to form a daisy chain, as shown in the Fig. 5.3.17. As shown in the Fig. 5.3.17, the interrupt request line ($\overline{INTR}$) is common to all devices and the interrupt acknowledge line (INTA) is connected in a daisy-chain fashion. In daisy chain fashion the signal is allowed to propagate serially through the devices. When more than one devices issue an interrupt request, the $\overline{INTR}$ line is activated and processor responds by setting the INTA line. This signal is received by device 1. Device 1 passes the signal to the device 2 only if it does require any service. If device 1 requires service, it blocks the INTA line and puts its identification code on the data lines. Therefore, in daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority.

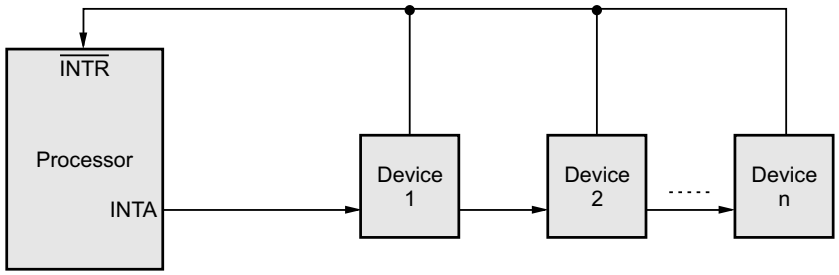


Fig. 5.3.17 Interrupt priority system using daisy chain

5.3.3.7 Pipeline Interrupts

When an interrupt occurs in a computer system, the computer CPU must be able to identify the interrupting instruction and the register contents needed for execution of interrupt service routine. This is not a problem when instructions are executed in sequence and only one instruction is executed at a time. However, in a pipelined processor several instructions are in process concurrently and it may happen that the instruction which was issued earlier finish its execution after execution of its successive instructions. This is known as out of sequence execution of instruction. Let us see the example shown in Fig. 5.3.18. The instruction issued first is a **multiply** instruction and it requires eight cycles for execution. The next two instructions issued after multiply instruction are **add** instructions. They require five cycles. If these instructions are processed by one or more pipelined units, two add instruction may complete before the multiply instruction as shown in Fig. 5.3.18 (a).

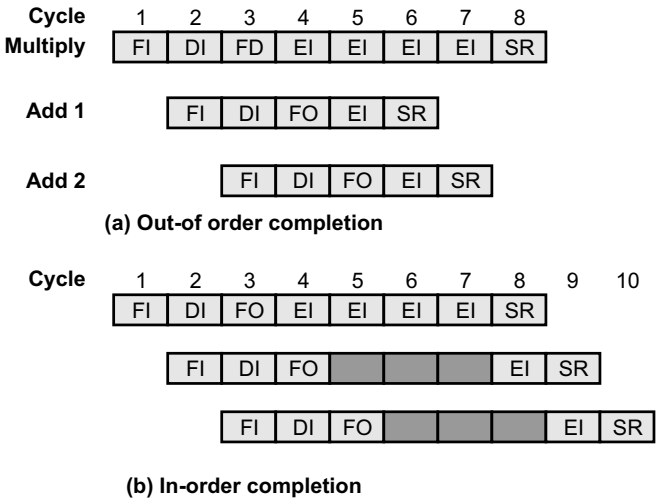

Cycle 1 2 3 4 5 6 7 8
Multiply FI DI FD EI EI EI EI SR
Add 1 FI DI FO EI SR
Add 2 FI DI FO EI SR
(a) Out-of order completion
Cycle 1 2 3 4 5 6 7 8 9 10
FI DI FO EI EI EI EI SR
FI DI FO EI SR
FI DI FO EI SR
(b) In-order completion

Fig. 5.3.18

Suppose, however, that add 1 instruction generates an interrupt due to overflow in its execution stage EI corresponding to cycle 5. Control must then be transferred to an interrupt routine designed to service adder overflow. Meanwhile, it is possible that the ongoing multiply instruction will generate another interrupt, say in cycle 7. This second interrupt can change the CPU's state (context) in ways that prevent proper processing of the first interrupt. In particular, registers affected by add1 interrupt can be further modified by the multiply interrupt routine so that proper execution of add1 interrupt subroutine may not be possible. In this situation the CPU state is imprecise. To avoid such imprecise CPU state or context we have to satisfy the conditions listed below :

- All instructions issued prior to instruction I must have completed their execution.

- No instruction has been issued after instruction I.
- The program counter should contain instructions address.

These conditions can be satisfied by delaying the add instructions so that they complete after multiply instruction as shown in Fig. 5.3.18 (b). However, such in-order execution method increases processing time for the three instructions from eight cycles to ten cycles. As a result, some of the performance benefit of pipelining is lost.

An alternative solution is to allow out-of-order completion but provide a mechanism to recover the precise state of context of the processor at the time of the interrupt. Usually, a small register buffer known as **history buffer (HB)** is introduced to store temporarily the initial state of every register. Hence if an interrupt occurs during any instruction execution or even if a second conflicting interrupt is generated the corresponding precise CPU state can be recovered from the values stored in HB.

5.3.3.8 Exceptions

An interrupt is an event that suspends the processing of currently executing program and begins the execution of another program. So far we have seen the interruption is activated by an I/O device in the form of a hardware signal ; however, many events can cause an interrupts. All such events are called exceptions. Therefore, an I/O interrupt is a subtype of exception. Apart from I/O interrupts the exceptions can be classified as faults, traps, or aborts depending on the way they are reported, and whether or not restart of the instruction causing the exception is supported.

Faults

Faults are the exceptions that are detected and serviced *before* the execution of the faulting instruction. For example, in virtual memory system if page or segment referenced by processor is not present, the operating system fetches the page or segment from disk using fault exception routine, and then 80386 restarts processing using referenced page or segment

Traps

Traps are exceptions that are reported immediately *after* the execution of the instructions which causes the problem. User defined interrupts are the examples of traps.

Aborts

Aborts are exceptions which do not permit the precise location of the instruction causing the exception to be determined. Aborts are used to report severe errors, such as hardware error, or illegal values in the system tables.

5.3.3.9 PCI Interrupts

The PCI bus has four interrupt request lines : $\overline{\text{INTA}}$, $\overline{\text{INTB}}$, $\overline{\text{INTC}}$ and $\overline{\text{INTD}}$. A single function I/O device with interrupt capability must use $\overline{\text{INTA}}$ as its interrupt request line whereas multifunction I/O devices can use all four lines. A particular pattern on the PCI bus's command lines denotes interrupt acknowledge.

Each PCI compatible device consists of CR register. The CR register consists of 8-bit **interrupt pin** register. It tells system controller which interrupt request line $\overline{\text{INTX}}$ the I/O device is using. A second 8-bit register in CR called the **interrupt line** register specifies the system controller's input line that is connected to $\overline{\text{INTX}}$ so that the routing of the interrupt request lines is programmable. The system controller can use this fact to determine the I/O device's interrupt-request priority and to access its interrupt vectors.

Review Questions

1. *Why are interrupt masks provided in any processor ?* **May-06, Marks 2**
2. *Explain the use of vectored interrupts in processors. Why is priority handling desired in interrupt controllers ? How do the different priority schemes work ?* **May-06, Marks 10**
3. *Discuss the general steps involved in interrupt driven data transfer.* **Dec.-06, Marks 6**
4. *How does the processor handle an interrupt request ?* **May-07, Marks 2**
5. *How do you connect multiple I/O devices to a processor using interrupts ? Explain with suitable diagrams.* **Dec.-07, Marks 8**
6. *Explain in detail about interrupt handling.* **Dec.-07, Marks 16**
7. *What is priority interrupt ?* **May-08, Marks 8**
8. *What is an non-maskable interrupt ? What is the action performed on receipt of a NMI ?* **Dec.-08, Marks 2**
9. *Write short note on : Interrupts* **Dec.-08, Marks 5**
10. *Explain about vectored interrupt with daisy chain arrangement.* **Dec.-08, Marks 6**
11. *What are vectored interrupts ?* **May-09, Marks 2**
12. *What are the types of interrupts ?* **May-09, Marks 2**
13. *Describe the various mechanism for accessing I/O devices.* **May-09, Marks 16**
14. *Explain why I/O devices cannot be connected directly to the systems bus.* **May-10, Marks 2**
15. *Define an interrupt.* **Dec.-10, Marks 2**
16. *Describe vectored interrupt scheme with a neat block diagram.* **Dec.-11, 12, Marks 8**
17. *List the functionalities of an interface.* **May-12, Marks 8**

5.4 Comparison between Programmed I/O and Interrupt Driven I/O

May-07, 08, 09

The Table 5.4.1 gives the comparison between programmed I/O and interrupt driven I/O.

Sr. No.	Programmed I/O	Interrupt driven I/O
1.	In programmed I/O, processor has to check each I/O device in sequence and in effect 'ask' each one if it needs communication with the processor. This checking is achieved by continuous polling cycle and hence processor can not execute other instructions in sequence.	External asynchronous input is used to tell the processor that I/O device needs its service and hence processor does not have to check whether I/O device needs it service or not.
2.	During polling processor is busy and therefore, have serious and decremental effect on system throughput.	In interrupt driven I/O, the processor is allowed to execute its instructions in sequence and only stop to service I/O device when it is told to do so by the device itself. This increases system throughput.
3.	It is implemented without interrupt hardware support.	It is implemented using interrupt hardware support.
4.	It does not depend on interrupt status.	Interrupt must be enabled to process interrupt driven I/O.
5.	It does not need initialization of stack.	It needs initialization of stack.
6.	System throughput decreases as number of I/O devices connected in the system increases.	System throughput does not depend on number of I/O devices connected in the system.

Table 5.4.1 Comparison between programmed I/O and interrupt driven I/O

Review Questions

1. *What is the advantage of using interrupt initiated data transfer over transfer under program control without interrupt ?* **May--07, Marks 2**
2. *Give the comparison between programmed I/O and interrupt driven I/O.*

5.5 Direct Memory Access (DMA)

May-07, 08, 09, 12, Dec.-03, 06, 07, 08, 10, 11, 12

In software control data transfer, processor executes a series of instructions to carry out data transfer. For each instruction execution fetch, decode and execute phases are required. Fig. 5.5.1 gives the flowchart to transfer data from memory to I/O device.

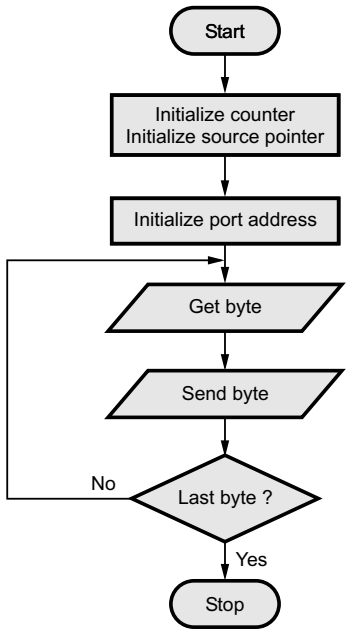


Fig. 5.5.1

Thus to carry out these tasks processor requires considerable time. So this method of data transfer is not suitable for large data transfers such as data transfer from magnetic disk or optical disk to memory. In such situations hardware controlled data transfer technique is used.

There are two main drawbacks in both the techniques, programmed I/O and interrupt driven I/O :

1. The I/O transfer rate is limited by the speed with which the CPU can test and service a device.
2. The time that the CPU spends testing I/O device status and executing a number of instructions for I/O data transfers can often be better spent on other processing tasks.

To overcome above drawbacks an alternative technique, hardware controlled data transfer can be used.

Sr. No.	I/O Program controlled transfer	DMA transfer
1.	It is software controlled data transfer.	Hardware controlled data transfer.
2.	Data transfer speed is low.	Data transfer speed is high.
3.	CPU is involved in the transfer.	CPU is not involved in the transfer.
4.	Extra hardware is not required.	DMA controller is required to carry-out data transfer.
5.	During data transfer data is routed through processor.	During data transfer data does not routed through processor.

Table 5.5.1 Comparison between I/O program controlled transfer and DMA transfer

5.5.1 Hardware Controlled Data Transfer

In this technique external device is used to control data transfer. External device generates address and control signals required to control data transfer and allows peripheral device to directly access memory. Hence this technique is referred to as **Direct Memory Access** (DMA) and external device which controls the data transfer is referred to as **DMA controller**.

Fig. 5.5.2 shows that how DMA controller operates in a computer system.

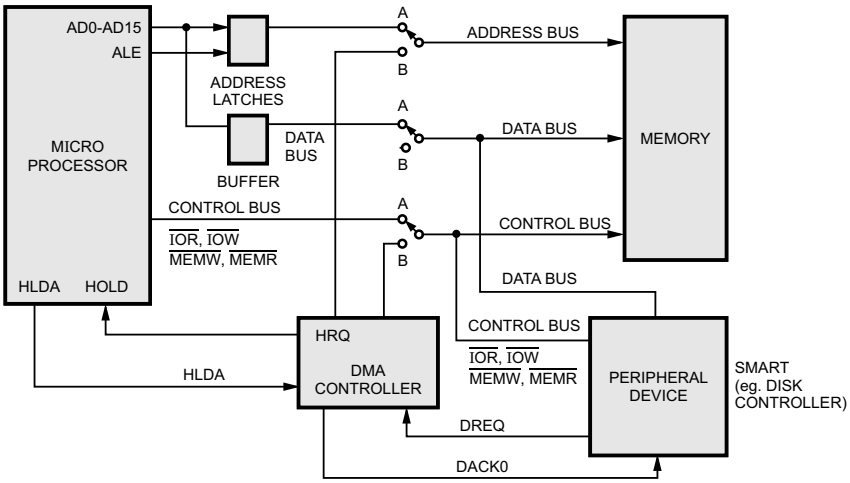


Fig. 5.5.2 DMA controller operating in a microprocessor system

5.5.2 DMA Idle Cycle

When the system is turned on, the switches are in the A position, so the buses are connected from the processor to the system memory and peripherals. Processor then executes the program until it needs to read a block of data from the disk. To read a block of data from the disk processor sends a series of commands to the disk controller device telling it to search and read the desired block of data from the disk. When disk controller is ready to transfer first byte of data from disk, it sends DMA request DRQ, signal to the DMA controller. Then DMA controller sends a hold request HRQ, signal to the processor HOLD input. The processor responds this HOLD signal by floating its buses and sending out a hold acknowledge signal HLDA, to the DMA controller. When the DMA controller receives the HLDA signal, it sends a control signal to change switch position from A to B. This disconnects the processor from the buses and connects DMA controller to the buses.

5.5.3 DMA Active Cycle

When DMA controller gets control of the buses, it sends the memory address where the first byte of data from the disk is to be written. It also sends a DMA acknowledge, DACK signal to the disk controller device telling it to get ready to output the byte. Finally, it asserts both the $\overline{IOR}$ and $\overline{MEMW}$ signals on the control bus. Asserting the $\overline{IOR}$ signal enables the disk controller to output the byte of data from the disk on the data bus and asserting the $\overline{MEMW}$ signal enables the addressed memory to accept data from the data bus. In this technique data is transferred directly from the disk controller to the memory location without passing through the processor or the DMA controller.

Thus, the CPU is involved only at the beginning and end of the transfer as shown in the Fig. 5.5.3. (See Fig. 5.5.3 on next page). The data transfer is monitored by DMA controller, which is also called **DMA channel**. When the CPU wishes to read or write a block of data, it issues a command to the DMA module or DMA channel by sending the following information to the DMA channel/controller :

1. A read or write operation.
2. The address of I/O device involved.
3. The starting address in memory to read from or write to.
4. The number of words to be read or written. (See Fig. 5.5.3 on next page.)

5.5.4 DMA Channels

For performing the above functions, the basic blocks required in a DMA channel/controller are shown in Fig. 5.5.4. DMA controller communicates with the CPU via the data bus and control lines. The registers in DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (Register select)

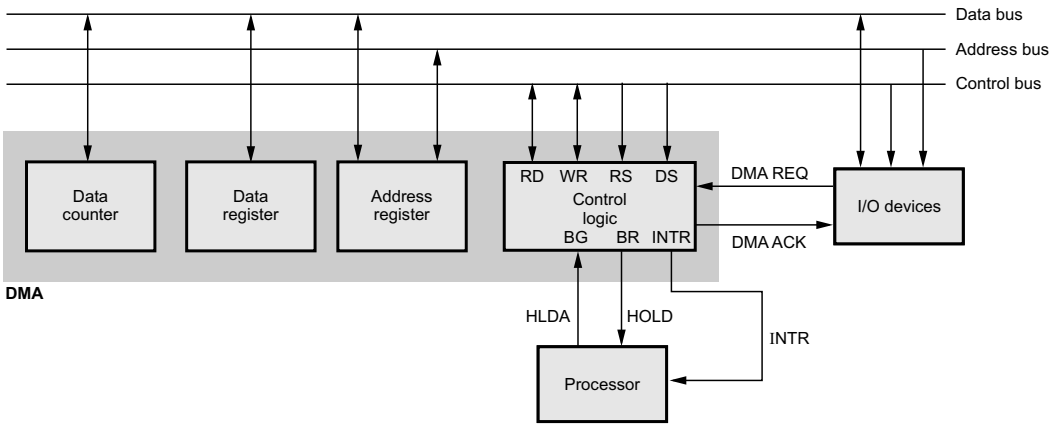


Fig. 5.5.4 Typical DMA block diagram

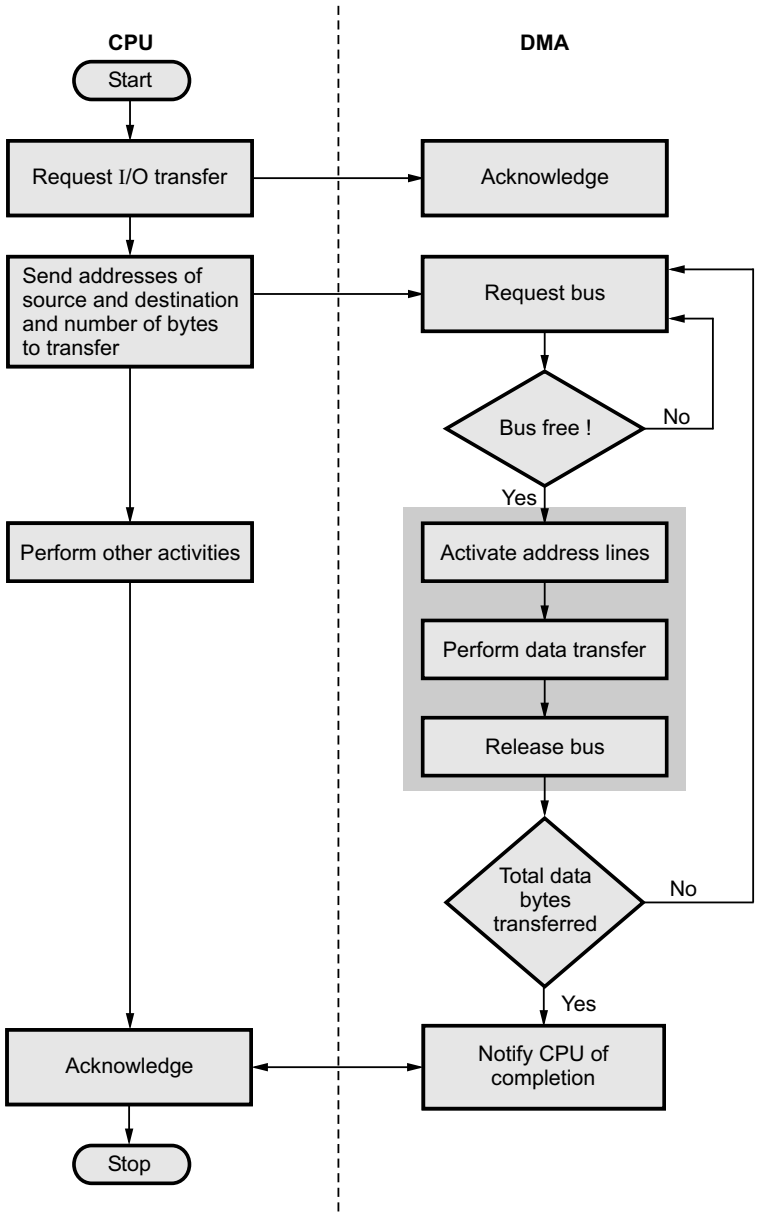


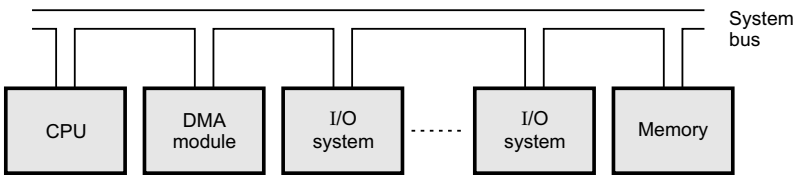
Fig. 5.5.3 Interaction of CPU and DMA

inputs. The RD (Read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write the DMA registers RD and WR signals are input signals for DMA. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the

RD or WR signals (RD and WR are now output signals for DMA). DMA consists of data count, data register, address register and control logic. Data counter register stores the number which gives the number data transfers to be done in one DMA cycle. It is automatically decremented after each word transfer. Data register acts as buffer whereas address register initially holds the starting address of the device. Actually, it stores the address of the next word to be transferred. It is automatically incremented or decremented after each word transfer. After each transfer data counter is tested for zero. When the data count reaches zero, the DMA transfer halts. The DMA controller is normally provided with an interrupts capability, in which case it sends an interrupt to processor to signal the end of the I/O data transfer.

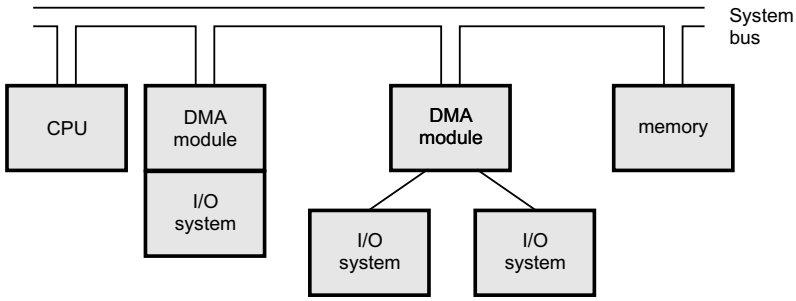
Fig. 5.5.5 gives the possible DMA configurations. As shown in Fig. 5.5.5 (a), all modules, CPU, DMA module, I/O system and memory share the same system bus. In system this programmed I/O technique is used. The data is exchanged between memory and I/O system through DMA module. Each transfer of a word consumes two bus cycles.

Fig. 5.5.5 (b) shows the another type of DMA configuration in which there is different path between DMA module and I/O system which does not include the system bus. The DMA logic may actually be a part of an I/O system, or it may be a separate module that controls one or more I/O systems.

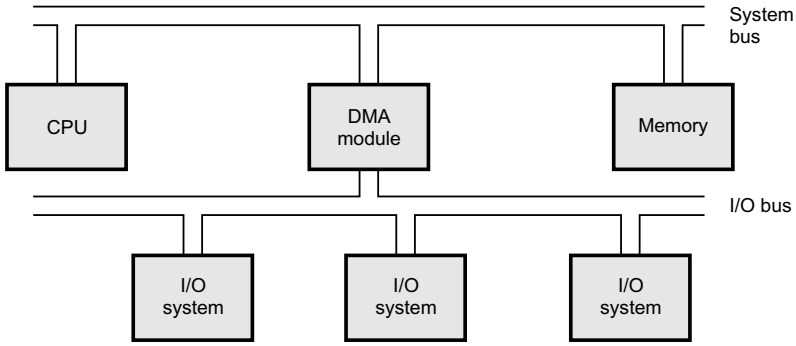


(a) Single bus, detached

A one step ahead is the third type of configuration as shown in Fig. 5.5.5 (c). In this I/O systems are connected to module using an I/O bus. This reduces the number of I/O interfaces in the DMA module to one and provides an easy expandable configuration.



(b) Single bus, integrated DMA-I/O



(c) Separate I/O bus

Fig. 5.5.5 DMA configurations

5.5.5 Data Transfer Modes

DMA controller transfers data in one of the following three modes :

- Single transfer mode (cycle stealing)
- Block transfer mode
- Demand or burst transfer mode

Single transfer mode

In this mode device can make only one transfer (byte or word). After each transfer DMAC gives the control of all buses to the processor. Due to this processor can have access to the buses on a regular basis.

It allows the DMAC to time share the buses with the processor, hence this mode is most commonly used.

The operation of the DMA in a single transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the processor.
3. The processor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts $\overline{\text{DACK}}$ to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts $\overline{\text{DACK}}$ line.
7. The word/byte transfer count is decremented and the memory address is incremented.
8. The HOLD line is deasserted to give control of all buses back to the processor.
9. HOLD signal is reasserted to request the use of buses when I/O device is ready to transfer another byte or word. The same process is then repeated until the last transfer.
10. When the transfer count is exhausted, terminal count is generated to indicate the end of the transfer.

Block transfer mode

In this mode device can make number of transfers as programmed in the word count register. After each transfer word count is decremented by 1 and the address is decremented or incremented by 1. The DMA transfer is continued until the word count "rolls over" from zero to FFFFH, a Terminal Count (TC) or an external END of Process ($\overline{\text{EOP}}$) is encountered. Block transfer mode is used when the DMAC needs to transfer a block of data.

The operation of DMA in block transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts $\overline{\text{DACK}}$ to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts $\overline{\text{DACK}}$ line.

7. The word/byte transfer count is decremented and the memory address is incremented.
8. When the transfer count is exhausted, the data transfer is not complete and the DMAC waits for another DMA request from the I/O device, indicating that it has another byte or word to transfer. When DMAC receives DMA request steps through are repeated.
9. If the transfer count is not exhausted, the data transfer is complete then DMAC deasserts the HOLD to tell the microprocessor that it no longer needs the buses.
10. Microprocessor then deasserts the HLDA signal to tell the DMAC that it has resumed control of the buses.

Demand Transfer mode

In this mode the device is programmed to continue making transfers until a TC or external $\overline{EOP}$ is encountered or until DREQ goes inactive.

The operation of DMA in demand transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts $\overline{DACK}$ to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts $\overline{DACK}$ line.
7. The word/byte transfer count is decremented and the memory address is incremented.
8. The DMAC continues to execute transfer cycles until the I/O device deasserts DRQ indicating its inability to continue delivering data. The DMAC deasserts HOLD signal, giving the buses back to microprocessor. It also deasserts $\overline{DACK}$.
9. I/O device can re-initiate demand transfer by reasserting DRQ signal.
10. Transfer continues in this way until the transfer count has been exhausted.

The flowcharts in the Fig. 5.5.6 summarized the three data transfer modes of DMA.

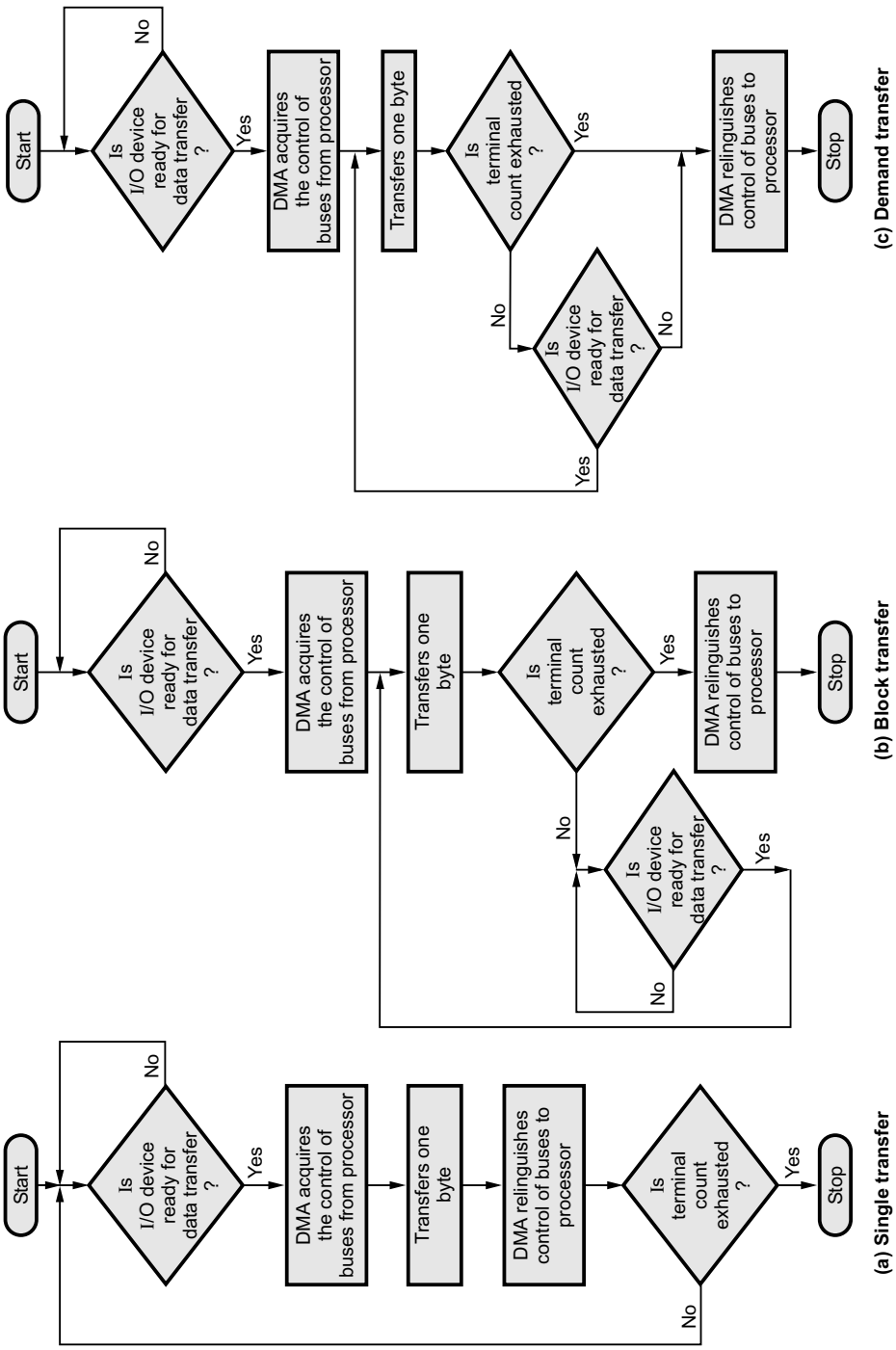


Fig. 5.5.6 Three data transfer modes of DMA

Example 5.5.1 *What are the different input and output signals of DMA controller ? Why are the read and write control signals are bidirectional ? Under what condition and for what purpose they are used as inputs and outputs ?* **May-07, 10 Marks 10**

Solution :

DMA controller signals

Signals	Input : I Output : O Input/Output : I/O
Data lines	I/O
$\overline{IOR}$	I/O
$\overline{IOW}$	I/O
A_0 - A_3	I/O
A_4 - A_7	O
Ready	I
HRQ	O
HLDA	I
$\overline{MEMR}$	O
$\overline{MEMW}$	O
AEN	O
ADSTB	O
TC	O
MARK	O
DRQ_0 - DRQ_3	I
$DACK_0$ - $DACK_3$	O

When CPU is master, the DMA read/write signals are input signal ; however when DMA is master it drives these signals, i.e. read/write signals are output signals. Therefore, in DMA read and write control signals are bidirectional.

Review Questions

1. *Explain with block diagram the DMA transfer in a computer system.* **Dec.-03, Marks 8**
2. *Describe DMA and it implementation. State the advantage of DMA over the other modes of I/O transfer.* **Dec.-08, Marks 5**
3. *What is DMA operation ? State its advantages.* **Dec.-06, Marks 2**
4. *Why do we need DMA ?* **Dec.-07, Marks 2**
5. *Write note on : DMA* **Dec.-07, Marks 4**
6. *Explain in detail about direct memory access methods (DMA).* **Dec.-08, Marks 10**
7. *Explain the steps involved in the DMA operation.* **May-09, Marks 16**
8. *Discuss the DMA driven data transfer technique.* **May-07, Marks 8**
9. *Explain the DMA mode of date transfer.* **Dec.-10, 12, Marks 16**
10. *What are the necessary operations needed to start an I/O operation using DMA ?* **Dec.-07, Marks 2**
11. *Describe the various mechanisms of accessing input output devices .* **May-09, Marks 16**
12. *Explain how DMA transfer is accomplished with a neat diagram.* **Dec.-06, Marks 10**
13. *Explain the use of DMA controllers in a computer system with a neat diagram.* **May-08, Marks 2**
14. *What is DMA ? Draw the block diagram and explain it in detail.* **Dec.-11, Marks 8**
15. *What is DMA ? Explain the different modes of operations with the working principle.* **May-12, Marks 16**

5.6 Interface Circuits

May-12

We know that, an I/O interface consists of circuits which connect an I/O device to a computer bus. As shown in Fig. 5.6.1 on one side of the interface we have the bus signals for address, data and control. On the other side we have a data path with its associated controls to transfer data between the interface and the I/O device. This side is called a **port**. The port can be classified as serial port or parallel port.

5.6.1 Parallel Port

Parallel port is used to send or receive data having group of bits (8-bits or 16-bits) simultaneously. According to usage, hardware and control signal requirements parallel ports are classified as input port and output port. Input ports are used to receive the data whereas output ports are used to send the data.

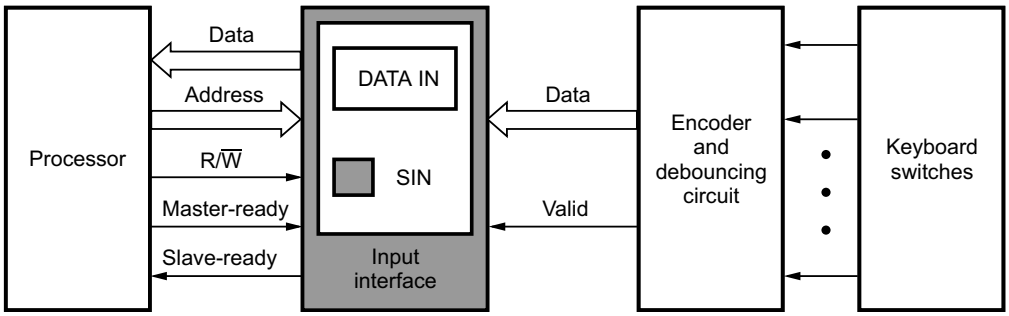


Fig. 5.6.1 Hardware components for connecting a keyboard to a processor

5.6.1.1 Input Port

Commonly used input device is a keyboard. Fig. 5.6.1 shows the hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open. When key is pressed, corresponding signal alters and encoder circuit generates ASCII code for the corresponding key.

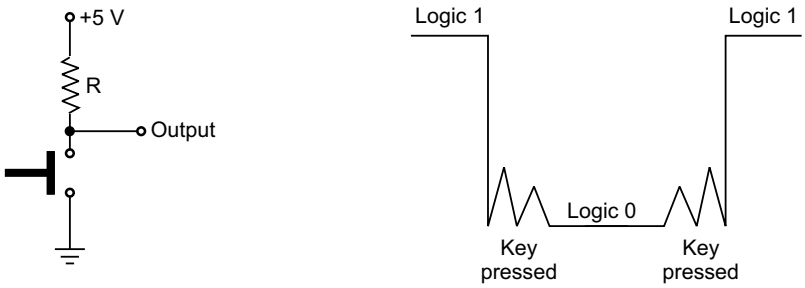


Fig. 5.6.2 Bouncing of key switch

For interfacing switches to the microprocessor based systems, usually push button keys are used. These push button keys when pressed, bounces a few times, closing and opening the contacts before providing a steady reading, as shown in the Fig. 5.6.2. Reading taken during bouncing period may be faulty. Therefore, microprocessor must wait until the key reach to a steady state; this is known as **key debounce**.

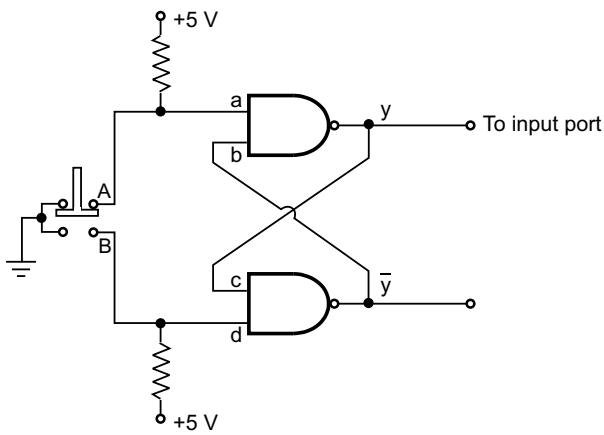


Fig. 5.6.3

Key-debouncing circuit included in the block diagram of Fig. 5.6.3 eliminate effect of key bouncing. The problem of key bounce can be eliminated using key debounce circuit (hardware approach) or software approach. When debouncing is implemented in software, the I/O routine that reads a ASCII code of character from the keyboard waits long enough to ensure that bouncing has subsided.

Fig. 5.6.3 shows the hardware approach to prevent key bouncing. It consists of flip-flop. The output of flip-flop shown in Fig. 5.6.3 is logic 1 when key is at position A (unpressed) and it is logic 0 when key is at position B, as shown in Table 5.6.1. It is important to note that, when key is in between A and B, output does not change, preventing bouncing of key output. In other words we can say that output does not change during transition period, eliminating key debouncing.

key position	a	b	Y	c	d	$\overline{Y}$
A	0	0	1	1	1	0
B	1	1	0	0	0	1
Between A and B	1	$\overline{Y}$	No change	Y	1	No change

Table 5.6.1

The output of encoder in Fig. 5.6.4 consists of the bits that represent the encoded character and one control signal called **valid**, which indicates that a key is being pressed. The encoder circuit sends this information to the interface circuit. The interface circuit consists of a data register, DATA IN, and a status flag, SIN. When key is pressed, the valid signal is activated, (changes from 0 to 1) causing the ASCII code to be loaded into DATA IN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATA IN register. The interface circuit is connected to an asynchronous bus on which transfers are controlled by using the handshake signals master-ready and slave-ready as explained in section 5.2.2.2.

The Fig. 5.6.4 shows the typical internal circuitry for input port. Here, we receive the data input from the keyboard input device. When the key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates ASCII code for the corresponding character.

The input port consists of data register, DATA IN, and status flag, SIN. When a key is pressed, the valid signal activates and causes the ASCII code to be loaded into DATA IN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATA IN register. An address decoder is used to select the input interface when the high-order 31 bits of an address corresponds to any of the addresses assigned to this interface. Address bit A_0 determines whether the status or the data register is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the slave-ready signal when either Read-status or Read-data is equal to 1.

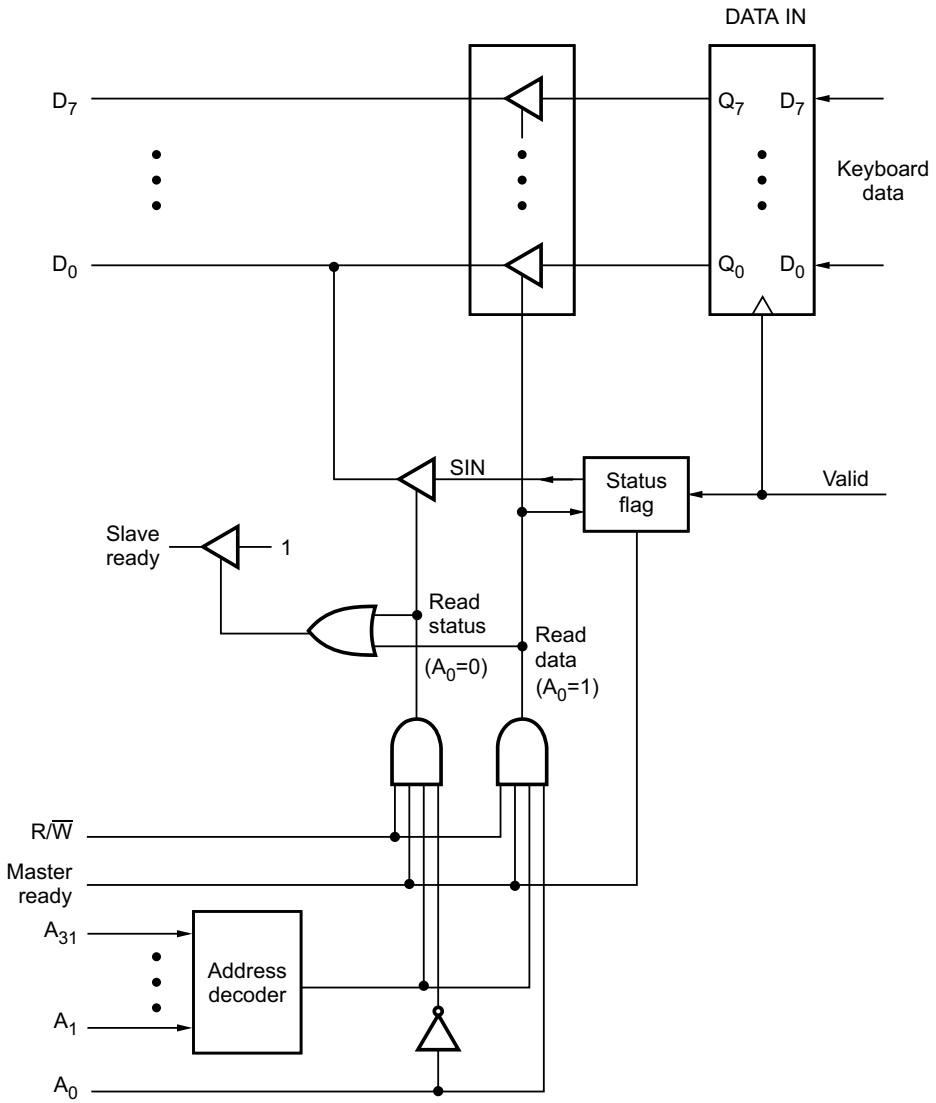


Fig. 5.6.4 Input interface circuit

5.6.1.2 Output Port

The Fig. 5.6.5 shows typical example of output port which is used to interface parallel printer.

The output port contains a data register, DATAOUT, and a status flag, SOUT. The SOUT flag is set to 1 when the printer is ready to accept another character, and it is cleared to 0 when a new character is loaded into DATAOUT by the processor. The Fig. 5.6.6 shows the detail internal circuit for output port.

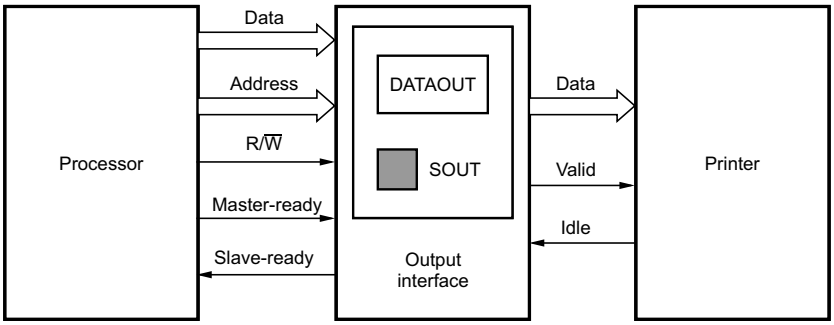


Fig. 5.6.5 Printer and processor interface

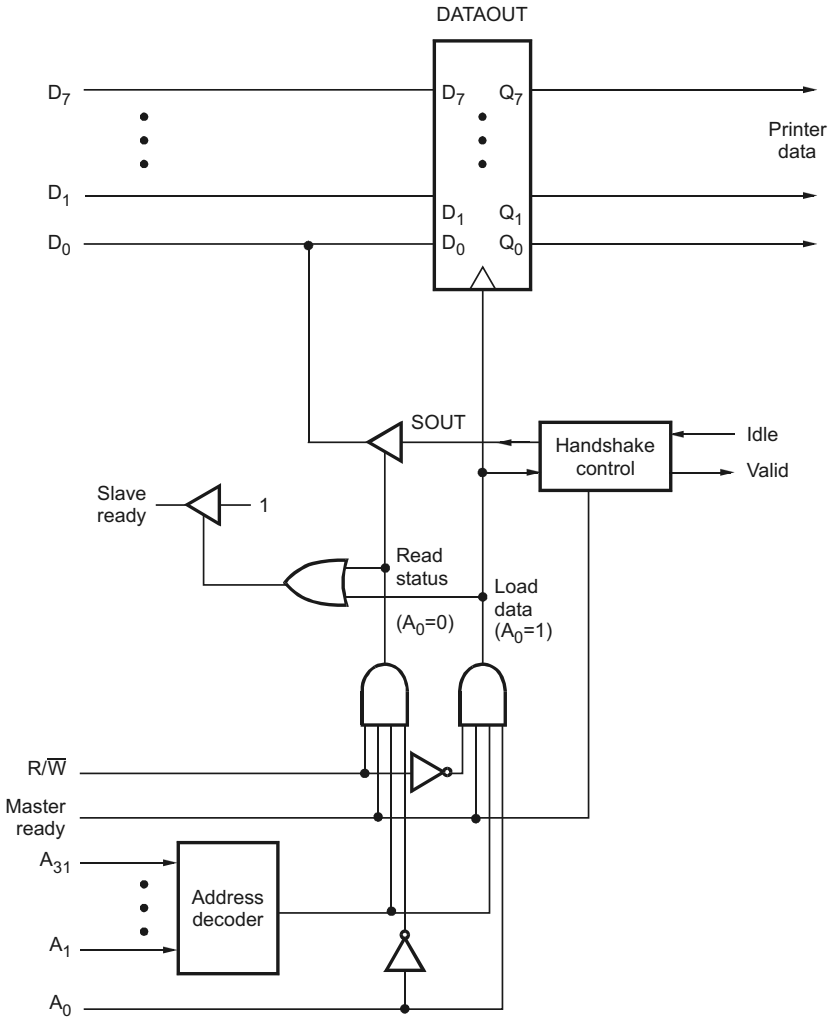


Fig. 5.6.6 Output interface circuit

5.6.1.3 Combined Input/Output Port

We can combine input and output interfaces as shown in the Fig. 5.6.7 . Here, the 30-bits of higher order address (A_{31} - A_2) are used to select overall interface. The low-order two bits of address (A_0 - A_1) are used to select one of the three addressable locations in the interface. These are : Two data registers and one status register. We know that flags SIN and SOUT are in the status register. We use labels RS_1 and RS_0 for inputs that determines the selection of desired register.

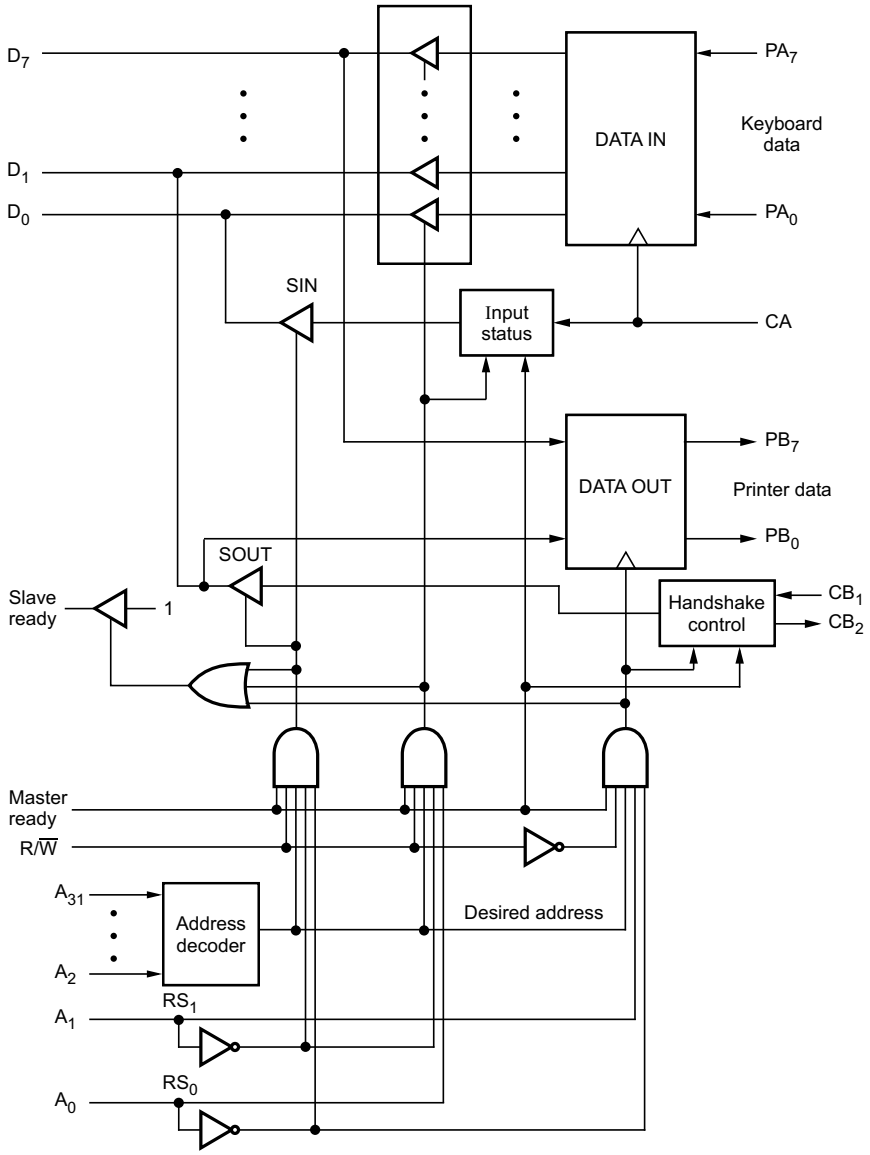


Fig. 5.6.7 Combined input/output interface circuit

5.6.1.4 Programmable Parallel Port

The input and output interfaces just described can be combined into a single interface and the direction of data flow can be controlled by data direction register. So single port can be programmed to use for input the data or output the data. Such a port is known as programmable parallel port. The Fig. 5.6.8 shows the simplified block diagram of 8-bit programmable parallel port. Here, data and port lines are bidirectional. There direction is controlled by data direction register (DDR). Two lines M_0 and M_1 are connected to the status and control. These two lines decides the mode of operation of the parallel port, i.e. whether to operate parallel port as a simple input/output or handshake input/output. Ready and Accept signals are provided as handshaking signals. The $\overline{INTR}$ signal is also provided to allow interrupt drives I/O data transfer.

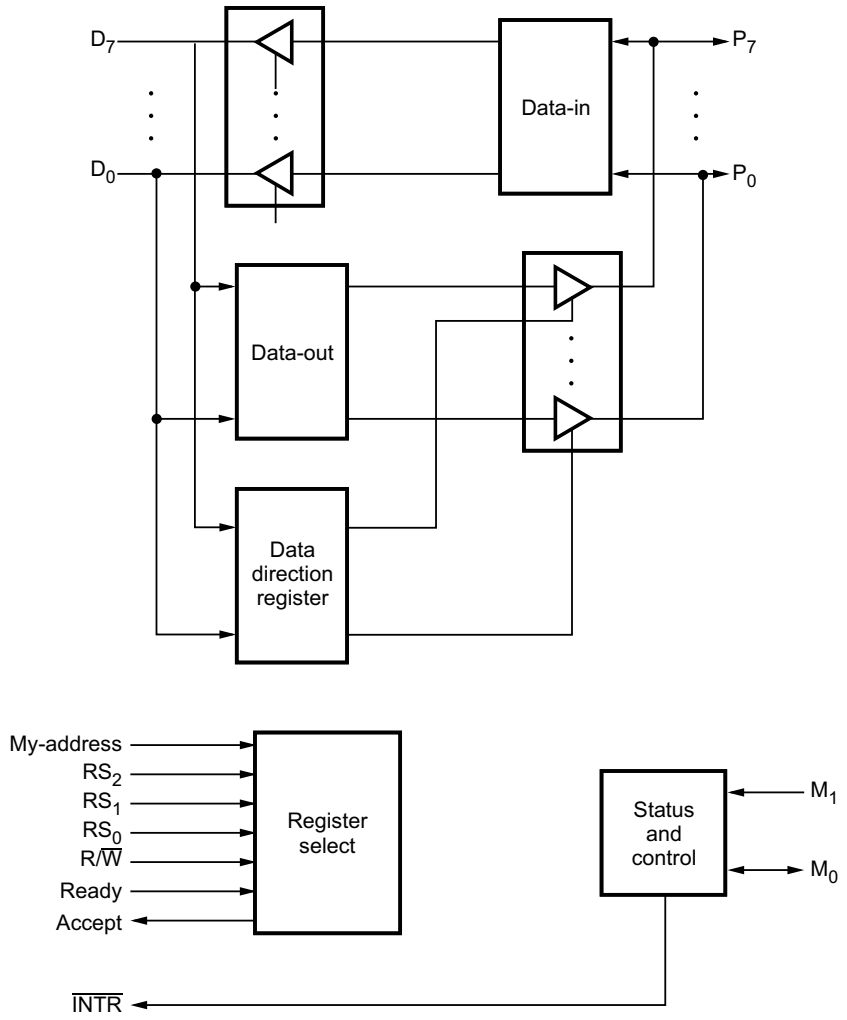


Fig. 5.6.8

5.6.2 Serial Port

A serial port is used to transmit / receive data serially, i.e., one bit at a time. A key feature of an interface circuit for a serial port is that it is capable of communicating in a bit serial fashion on the device side and in a bit parallel fashion on the processor side. A shift register is used to transform information between the parallel and serial formats. The Fig. 5.6.9 shows the block diagram of typical internal circuit for serial port.

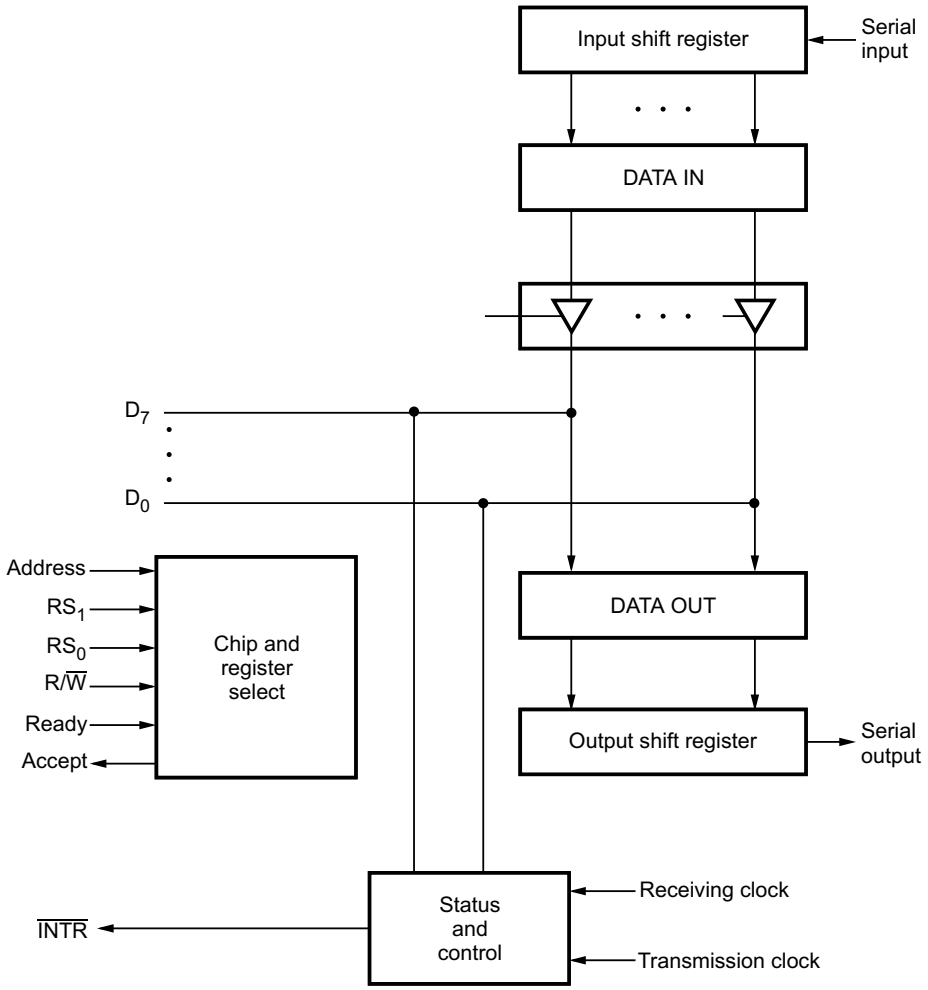


Fig. 5.6.9 A serial interface

As shown in the Fig. 5.6.9, the input shift register accepts serial data bit by bit and converts it into the parallel data. The converted parallel data is loaded in the data register and it is then read by the processor using data bus. When it is necessary to send

data serially. The data is loaded into DATAOUT register. It is then loaded into output shift register. Output shift register converts this parallel data into serial data.

5.6.3 Comparison between Serial and Parallel Interface

Most of the microprocessors are designed for parallel interface. In parallel interface number of lines required to transfer data depend on the number of bits to be transferred. For example, to transfer a byte of data, 8 lines are required and all 8 bits are transferred simultaneously. Thus for transmitting data over a long distance, using parallel interface is impractical due to the increase in cost of cabling. Parallel interface is also not practical for devices such as cassette tapes or a CRT terminal. In such situations, serial interface is used. In serial interface one bit is transferred at a time over a single line.

Sr. No.	Serial interface	Parallel interface
1.	It transfer data one bit at a time.	It can transmit more than one data bit at a time.
2.	Lower data transfer rate.	Faster data transfer rate.
3.	Needs less number of wires to connect devices in the system.	Needs more number of wires to connect devices in the system.
4.	Well suited for long distances, because fewer wires are used as compared to a parallel bus.	The interconnection penalty increases as distances increase. Thus not suitable for long distance interface.

Table 5.6.2 Comparison between serial and parallel interface

Review Questions

1. *Explain the general features of interfacing a parallel I/O port to a processor.*
2. *With a block diagram, explain, how a keyboard is connected to a processor ?*
3. *Draw a combined input/output interface circuit and explain the different operations clearly.*
4. *Explain serial port and a serial interface.*
5. *With a neat block diagram, explain a general 8-bit parallel interface circuit.*
6. *Draw and explain the block diagram of a typical serial interface. How does it compare with a parallel interface?*
7. *With a neat sketch, explain the individual input and output interface circuits. Also elicit their salient features.*
8. *What are the interface circuits ? Explain a general 8-bit parallel interface, with a neat diagram.*
9. *Explain the working of the parallel port with the necessary circuit diagram.* **May-12, Marks 16**

5.7 I/O Channel

So for we have seen various I/O techniques. These are the evolutionary steps in the I/O function . Let us summarizes these evolutionary steps.

1. In a simple microprocessor-controlled devices, a peripheral device is directly controlled by CPU.
2. A controller is then added to CPU to control peripheral devices with programming facility.
3. Now interrupts are employed in the configuration mentioned in step 2. This saves the CPU time which was required for a polling of I/O device.
4. DMA controller is introduced to give direct access to memory for I/O module.
5. The I/O module is then enhanced to become a processor with a specialized instruction set tailored for I/O. The I/O processor is capable of executing an I/O program in memory with directions given by CPU. It can execute I/O program without intervention of CPU.
6. The I/O module is further enhanced to have its own local memory. This makes it possible to control a large set of I/O devices with minimal CPU involvement.

   In step 5 and step 6 we have seen that the I/O module is capable of executing programs. Such I/O module is commonly known as **I/O channel**.

5.7.1 Characteristics of I/O Channel

An I/O channel has a special-purpose processor. This processor has an ability to execute I/O instructions and it can have complete control over I/O operation. The I/O instructions are stored in main memory. When I/O transfer is required, the CPU initiates an I/O transfer by instructing the I/O channel to execute an I/O program stored in the main memory. The I/O program specifies the device or devices, the area of memory storage, priority and actions to be taken for certain error conditions.

5.7.2 Types of I/O Channels

There are two main types of I/O channels.

- Selector channel and
- Multiplexer channel

5.7.2.1 Selector Channel

A selector channel controls multiple high-speed devices. However, at a time it can control the data transfer of only on device. It selects one device at a time and does the data transfer. Each device, or a small set of devices, is handled by a controller, or I/O module as shown in Fig. 5.7.1.

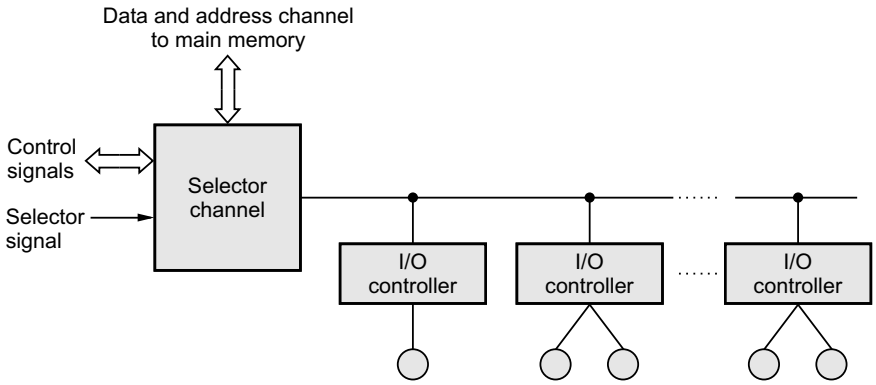


Fig. 5.7.1 Selector channel

5.7.2.2 Multiplexer Channel

A multiplexer channel can handle I/O with multiple devices at the same time. The multiplexer channel does the time multiplexing and communicates with each I/O controller in a allotted time slot. In multiplexer channel different devices can have different speeds. However, it is best suited for low speed devices.

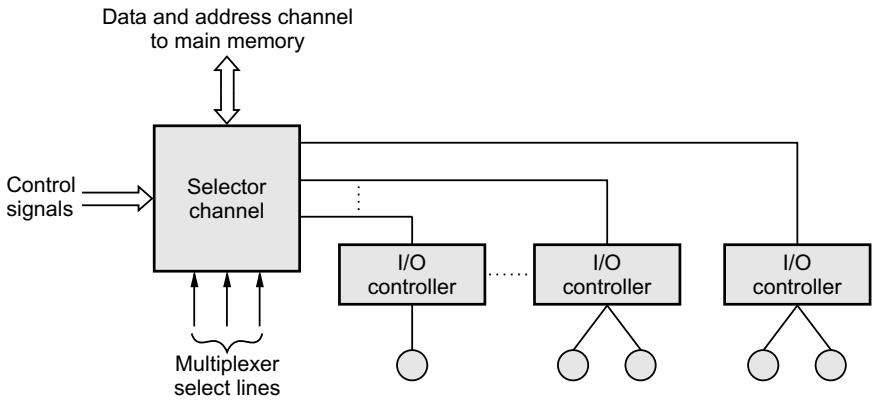


Fig. 5.7.2 Multiplexer channel

5.8 I/O Processor

Dec.-03

The I/O processor (IOP) has an ability to execute I/O instructions and it can have complete control over I/O operation. The I/O instructions are stored in main memory. When I/O transfer is required, the CPU initiates an I/O transfer by instructing the I/O channel to execute an I/O program stored in the main memory. The I/O program specifies the device or devices, the area of memory storage, priority and actions to be taken for certain error conditions.

5.8.1 Features and Functions of IOP

1. An IOP can fetch and execute its own instructions.
2. Instructions are specially designed for I/O processing.
3. In addition to data transfer, IOP can perform arithmetic and logic operations, branches, searching and translation.
4. IOP does all work involved in I/O transfer including device setup, programmed I/O, and DMA operation.
5. IOP can transfer data from an 8-bit source to 16-bit destination and vice versa.
6. Communication between IOP and CPU is through memory based control blocks. CPU defines tasks in the control blocks to locate a program sequence, called a channel program.
7. IOP supports multiprocessing environment. IOP and CPU can do processing simultaneously. This distributed processing approach improves system performance and flexibility.

5.8.2 Block Diagram of IOP

The Fig. 5.8.1 shows the block diagram of computer system with an I/O processor. The CPU and I/O processor work independently and communicate with each other using centrally located memory and DMA. The CPU does the processing of needed in the solution of computational tasks and IOP does the data transfer between various peripheral devices and the memory unit.

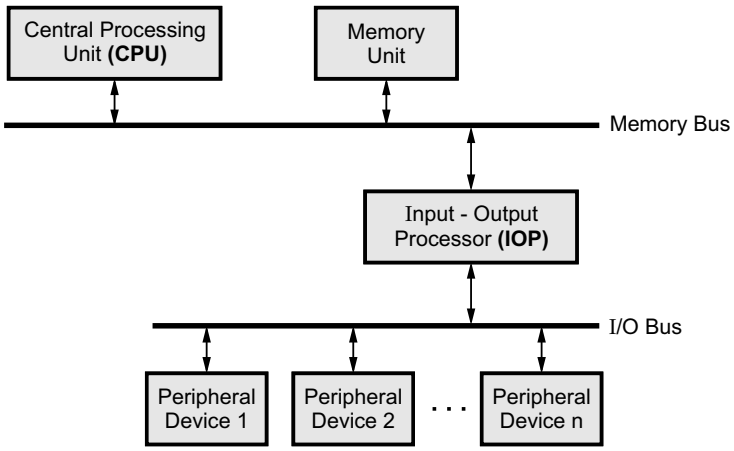


Fig. 5.8.1 Block diagram of a computer with I/O processor

5.8.3 CPU and IOP Communication

The communication between CPU and IOP may be different for different processor and IOP configurations. However, in most of cases the memory based control blocks are used to store the information about the task to be performed. The processor uses these blocks to leave information in it for the other processor. The memory control block are linked, i.e. the address of the next memory based control blocks is available in the previous memory based control block.

The Fig. 5.8.2 shows the flowchart of sequence of operations that are carried out during the CPU and IOP communication.

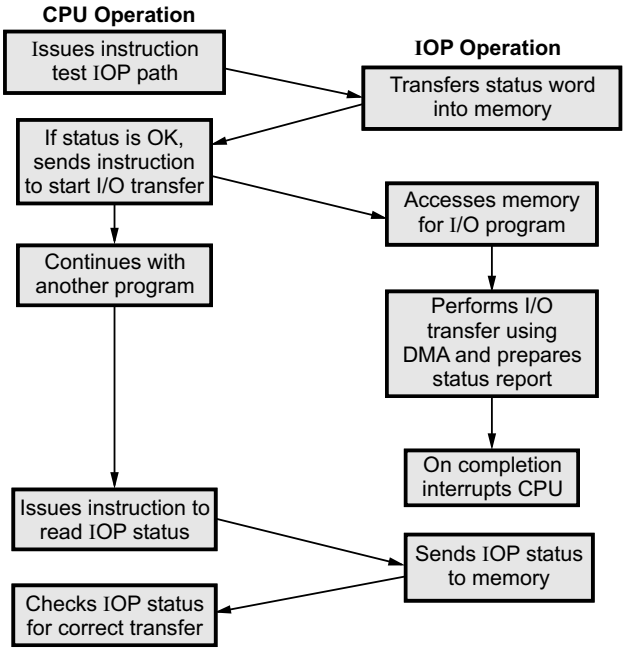

CPU Operation
Issues instruction test IOP path
IOP Operation
Transfers status word into memory
If status is OK, sends instruction to start I/O transfer
Accesses memory for I/O program
Continues with another program
Performs I/O transfer using DMA and prepares status report
On completion interrupts CPU
Issues instruction to read IOP status
Sends IOP status to memory
Checks IOP status for correct transfer

Fig. 5.8.2 CPU and IOP communication

The sequence of operations carried out during CPU and IOP communication are :

1. CPU checks the existance of I/O path by sending an instruction.
2. In response to this IOP puts the status word in the memory stating the condition of IOP and I/O device (busy, ready, etc.).
3. CPU checks the status word and if all conditions are OK, it sends the instruction to start I/O transfer along with the memory address where the IOP program is stored.
4. After this CPU continues with another program.
5. IOP now conducts the I/O transfer using DMA and prepares status report.
6. On completion of I/O transfer, IOP sends an interrupt request to the CPU.
7. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The status indicates whether the transfer has been completed or if any errors occurred during the transfer.

Review Questions

1. *Write short note on I/O processor.*
2. *What is the function IOP ?*
3. *Explain IOP with block diagram.*
4. *Explain the process of communication between a CPU and IOP.*

5.9 Operating Systems

An *Operating System* is a program which acts as an interface between a user of a computer and computer hardware, also provides an environment in which a user may execute programs.

Besides, providing program execution environment it also provides services like :

- Memory Management
- I/O Device Management
- File System Support
- Job Scheduling
- Swapping of other programs etc.

In another way we can define, Operating System Software (OS) is a class of programs which manage the resources of the computer, process commands, efficiently utilizes the computer hardware, and controls program execution.

5.9.1 What is an Operating System ?

Operating System is an important part of almost every computer system. It acts as a co-ordinator for every interaction activity between the users and their computer system.

A computer system can be roughly divided into 4 components.

- Hardware (CPU, memory, I/O devices)
- Operating System
- Application Programs (compilers, database systems, games)
- Users (people, other computers)

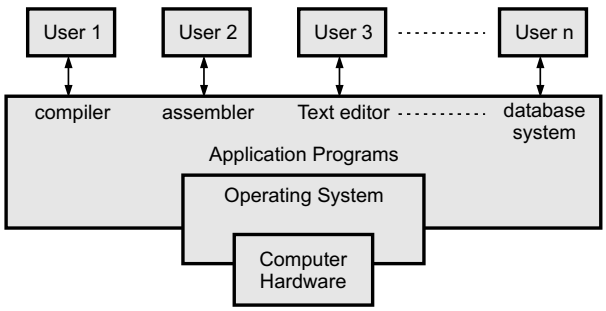


Fig. 5.9.1 Abstract view of the components of a computer system

The Fig. 5.9.1 shows, the operating system is a interface between the computer hardware and users. A user can communicate with the operating system using any of the two basic approaches :

1. Using Application Programs : Application programs invokes service of the operating systems by means of operating *system calls*.

2. Using Command Language : Operating System provides users with command language and the *command line interpreter* which an interpret the user command to give him what he wants.

5.9.2 Necessity of Operating System

Though operating system acts as a control program, some basic needs for which user requires it are :

1. To make the computer system more convenient to use. Without an operating system environment users cannot write or run their own programs. Operating system provides the basic platform over which users can run their own application.
2. To use computer system more efficiently. This efficiency is normally required for large multi-user systems, where number of users simultaneously accessing the resources. Operating system in this case, governs the activities and schedules the user processes properly, so that each of the users gets similar type of service.

We want some kind of interface which can control and co-ordinate the use of hardware resources by users of the computer system. The operating system provides that control interface.

5.9.3 Functions of Operating System

Any operating system, irrespective of type of computer on which being used plays three basic roles.

1. As a *`manager'* or *`resource allocator'*. A computer system has many resources (hardware and software) which may be required to solve, problem : CPU time, memory space, file storage, space, I/O device and so on. The OS allocates these resources to specific program and users as necessary for their tasks.
2. As a *`control program'*: A control program controls the execution of user programs to prevent errors and improper use of the computer.
3. As a *`command processor'*: The operating system reads commands from the user, verifies their accuracy, and execute the command given by the user.

5.9.4 Types of Operating Systems

5.9.4.1 Batch Operating System

To execute a particular task, the operating system has to load the resources necessary to execute the task. In the initial stages of operating system such time called **job setup time** was substantially large. To reduce job setup time, the task requiring same resources were grouped and executed sequentially. Suppose three programmers want to share the CPU time. First programmers job is Fortran programming, seconds job is Cobol

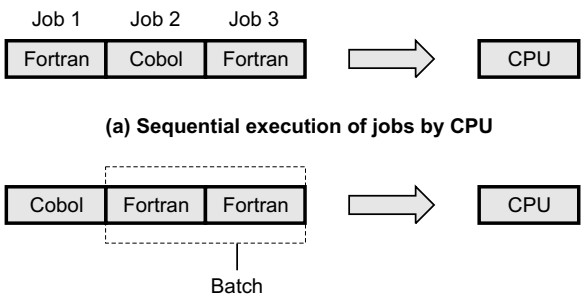


Fig. 5.9.2

programming and third programmers job is again Fortran programming. If all these programs executed sequentially then there would be more time required because of repetition of work (loading the Fortran compiler twice) which would increase the setup time. So the jobs with similar needs were batched together and run through the computer as a group which is shown in Fig. 5.9.2.

In the Fig. 5.9.2, we can see that if we process the jobs batchwise then we need to load the Cobol compiler one time and Fortran compiler one time. As first Fortran programmer will only load Fortran compiler while other has to just run his program, so less setup time will be required.

But sometimes the job is in a batch may give error, execution get stopped and CPU will remain idle. So to avoid this problem a automatic job sequencer was introduced. A small program, called a *resident monitor*, was developed for this purpose. The resident monitor is always resident in memory of computer system. This resident monitor transfers control from one job to another automatically which is shown in Fig. 5.9.3.

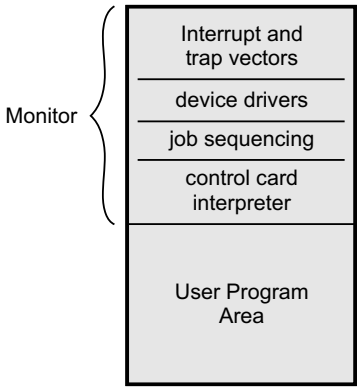


Fig. 5.9.3 Memory layout for a resident monitor

5.9.4.2 Multiprogramming Operating System

It is an attempt to increase CPU utilization by always having something for the CPU to execute.

The operating system picks one of the job in the job pool and begins to execute it. Eventually, the job may have to wait for an I/O operation to complete. In a multiprogramming system, the operating system will simply switch to another job and execute it. In turn, when the job wants to wait for an I/O, the system gets switched to another ready to execute job, and so on. Eventually, the first job will have finished waiting and will get the CPU back. As long as there is some job to execute, the CPU will never be idle, which is shown in Fig. 5.9.4.

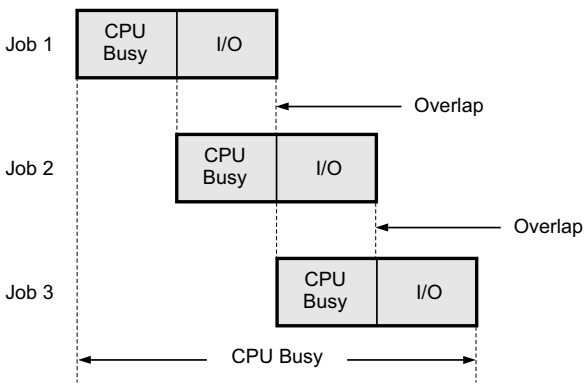


Fig. 5.9.4 Multiprogramming operating system

In multiprogrammed operating system several jobs ready to run are kept simultaneously in memory. As several programs are in memory at the same time, requires some form of memory management which is shown in Fig. 5.9.5.

As several jobs are ready to run at the same time so which job to be run first, second so on. For this purpose job scheduling is needed. Also multiprogramming system provides device scheduling.

Multiprogramming operating system usually support multiple users, in this case they are also called multi-user system. Multiuser systems provide facilities for maintenance of individual user environments, require user authentication for security and protection and provide per user resource usage accounting.

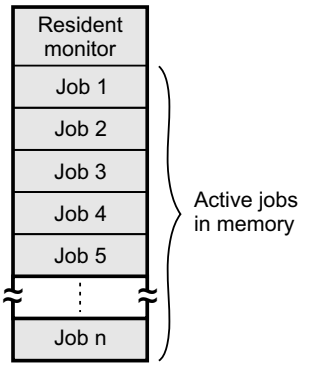


Fig. 5.9.5 Memory layout for a multiprogramming system

5.9.4.3 Time Sharing Operating System

A time sharing operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared computer. Each user has a separate program in memory. When a program executes, it typically executes for only a short time before it either finishes or needs to perform I/O which is shown in Fig. 5.9.6. A time-shared operating system allows the many users to simultaneously share the computer.

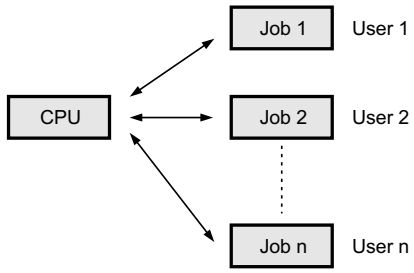


Fig. 5.9.6 Time-shared system

5.9.4.4 Real-Time Operating Systems

A real-time system is mostly used in control scientific experiments, industrial control systems etc. Real-time operating systems are used in environments where a large number of events, mostly external to the computer system must be accepted and processed in a short time or within certain deadlines.

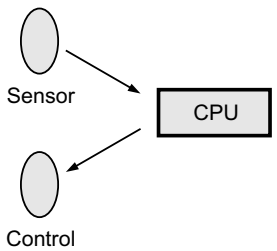


Fig. 5.9.7 Real-time operating system

The above Fig. 5.9.7 shows that sensors bring the data as input to computer. The computer must analyze the data and possibly adjust the controls to modify the sensor inputs; processing must be done within the defined fixed time period otherwise the system will fail. A real-time operating system provides time-critical device management, sophisticated forms of interrupt management, I/O buffering and speedy access.

5.9.5 Protection

The operating system allows sharing of resources. This sharing improves resource utilization. However, it increases the problems in the error handling. With sharing, many jobs could be adversely affected by a error in one program. Even more serious errors can occur in multiprogramming system where one program might modify the program

or data of another program or even the resident monitor which is used for automatic job sequencing.

Many programming errors are detected by the hardware. These errors are normally handled by the resident monitor. If a user program fails in some way, such as an illegal instruction or a memory addressing fault the hardware will trap the resident monitor. A trap is an internally generated interrupt, usually the result of a program error.

This approach works fine as long as the error is detected by the hardware. We must be sure, however, that all errors are detected. We must protect the operating system and other programs and their data from any malfunctioning program. Also protection needed for any shared resource. Thus at least three protections are needed :

- I/O protection
- Memory protection
- CPU protection

5.9.5.1 I/O Protection

There may be occasions when programmer decides not to use the built-in device drives and write his own. This approach will not provide proper service. So this has to be stopped.

We want the computer to allow the operating system to read the control cards, but prevent any user program from reading control cards. Thus we want the computer to behave differently for user programs and the operating system. We want two separate modes of operation : *user mode* and *supervisor mode* or system mode. A bit is added to the hardware of the computer to indicate the current mode : supervisor (0) or user (1).

We then define all I/O instructions to be privileged instructions. The hardware allows privileged instructions to be executed only in supervisor mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute it, but as it is an illegal instruction it traps to resident monitor which in turns protect I/O devices.

5.9.5.2 Memory Protection

In most of the cases, both the interrupt vector and the resident monitor occupy low memory, while user programs are loaded into high memory. Memory protection can then be a simple hardware fence register which separates memory into two parts user and monitor, shown in Fig. 5.9.8.

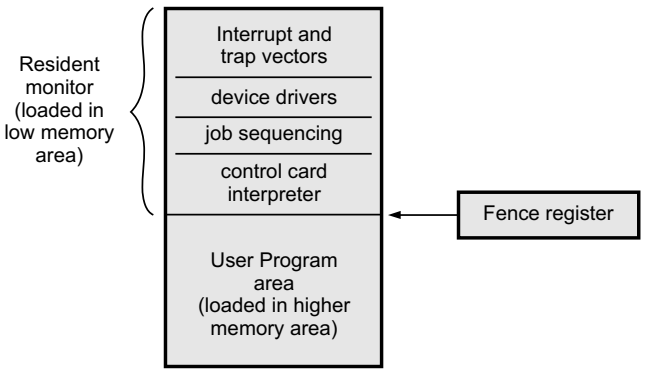

Resident monitor (loaded in low memory area)
Interrupt and trap vectors
device drivers
job sequencing
control card interpreter
Fence register
User Program area (loaded in higher memory area)

Fig. 5.9.8 Hardware address protection for a resident monitor

Monitor memory must not be accessed or modified in user mode. This protection is accomplished by comparing every address generated in user mode with fence register. Any attempt by a program executing in user mode to access monitor memory results in a trap to the monitor, which gives a **fatal error**. This prevent the user program from modifying the code or data structure of resident monitor which in turns protects memory.

5.9.5.3 CPU Protection

An CPU protection a small change in Architecture has been done by addition of a timer. This timer can be used which prevents a user program from getting stuck in an indefinite loop, and never returning control to the monitor. A timer can be set to interrupt the computer after some period of time. The period may be fixed or variable. A variable timer is implemented by a fixed rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches zero, an interrupt occurs.

Before turning over control to the user, the monitor makes sure that the timer is set to interrupt. If the timer interrupts control transfers automatically to the monitor which may treat it as a fatal error or decide to give the program more time which in turns protect CPU.

Demand Paging : Most virtual memory systems use a technique called **paging**. On any computer, there exists a set of memory addresses that programs can produce. Addresses can be generated using indexing base registers, segment registers and other ways. The program generated addresses are called **virtual addresses**.

Processes are started with none of their pages in memory. As soon as CPU tries to fetch the first instruction, it gets a page fault. So operating system will bring the page containing the first instruction. After some time processes has many pages it needs and settles down to run with relatively few page faults so to avoid this a new strategy is used called demand paging. So in **demand paging** pages are loaded only on demand, not in advance.

5.9.6 Distributed Operating System

These systems are evolved from the computer networks. In a network, multiple independent computer systems can communicate, sending files and information between them. The distributed operating system is a collection of individual computer systems capable of communication and co-operation via there hardware and software interconnection.

In distributed system the computations are distributed among several individual computer systems. The computer systems are communicate each other through various communication lines or telephone lines. The major reasons for building distributed systems are resource sharing, computation speed up, reliability and communication.

5.9.7 Operating System Services

Types of Services

As operating system provides environment for execution of users programs, it also provides certain services to programs and to the users of those programs. These services differ from one operating system to another, but there may be some common classes of services also. These services are provided for convenience of the programmer to make the programming task easier.

Program Execution

As users wants to execute their program. The system must be able to load a program into memory and run it. The program must be able to end its execution, either normally or abnormally. Loading the program and controlling it is done by operating system.

Input / Output Operations

A running program may requires input and output. This I/O may involve blank the screen on a CRT or print a file to some output device. So user program cannot execute I/O operations directly, the operating system must provide some means to do so.

File System Manipulation

A file can be considered as a logical storage unit on the media, when ever user wants to store or read his files from secondary storage medium operating system will use the concept of file system.

Error Detection

The operating system constantly needs to be aware of possible errors. These errors may occur in CPU and memory such as memory error or power failure or in I/O devices such as parity error on tape, printer out of paper or in the user program such as an arithmetic overflow, an attempt to access illegal memory location. For each type of error, the operating system takes the appropriate action to ensure the correct and consistent computing.

Resource Allocation

When there are multiple users or multiple job running at the same time, resources must be allocated to each of them. Many different types of resources are CPU time, main memory and file storage space of I/O devices etc. If more than one processes asking for same resources at the same time then some resource made to wait till that process relieve the resource. Some OS has the provision for sharing the resources.

Accounting

We want to keep track of which users use how much and what kinds of computer resources. This record-keeping may be for the purpose of paying for the system and its operation or simply for accounting usage statistics. Under this, at what time a particular user logged, which are the resources he has used and similar kind of activities come under record-keeping.

Protection

The owners of information stored in a multiuser computer system may want to control it's use means, they want to restrict its use. When several jobs are being executed simultaneously it should not be possible for one job to interface with the others. So conflicting demands for various resources need to be reconciled fairly and scheduled reasonably.

5.9.8 System Calls

Operating system services are provided in many different ways. Basic methods of providing services are system calls and system programs. System calls provide the interface between a running program and the operating system. These calls are generally available as assembly language instructions. System calls are roughly grouped into three categories : Process or job control, device and file manipulation, and information maintenance.

5.9.8.1 Process and Job Control

Operating system uses system calls to control the execution of running programs.

- End, Abort
- Load, Execute
- Create process, Terminate process
- Get process attributes, Set process attributes.
- Wait for time
- Wait event, Signal event.

A running program needs to be able to half its execution either normally or abnormally. A process or job executing one program may want to load and execute another program.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program and effectively have created a mechanism for one program to call another program. If we want to run both program concurrently we may need create process system call.

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including its priority, its maximum allowable execution time and so on. We may also want to terminate a job or process that we created if we find that it is incorrect or no longer needed.

Having created new jobs or processes, we may need to wait for them to finish execution. We may want to wait for a certain amount of time but more likely we want to wait for a specific event. The job or processes should then signal when that event has occurred.

5.9.8.2 File Manipulation

The system calls used for manipulation of files are :

- Create file, Delete file
- Open, Close
- Read, Write, Reposition
- Get file attributes, Set file attributes

We wants to create or delete a file. While doing this system calls requires the name of the file and perhaps some of its attributes. Once the file is created, we need to open it and use it. We may also read, write and reposition the file. Finally, we need to close the file. The various file attributes include the file name, a file type, protection codes, accounting information, date of file creation, file lastly updated, file size etc.

5.9.8.3 Device Manipulation

The system calls used for Device Management are-

- Request Device, Release Device
- Read, Write, Reposition
- Get Device Attributes, Set Device Attributes

Files can be thought of an abstract or virtual devices. Thus many of the system calls for files are needed for devices.

In multiuser system we must first request the device, to ensure that we have exclusive use of it. After we finished our work we must release it.

Once the device has been requested, we can read, write and reposition the device, just as with files.

5.9.8.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system.

- Get Time or Date, Set Time or Date
- Get System Data, Set System Data
- Get process, file, or Device Attribute, Set process.

Most systems have a system call to return the current time and date. Other system calls may return information about the system such as the number of current users, the version number of the operating system, the amount of free memory or disk space and so on.

5.9.9 System Call Implementation

System calls occur in different ways, depending upon the computer in use. On the IBM 360/370, there is a special instruction which traps directly to the operating system. The lower eight bits of the system call specify a number that identifies the type of call being made.

Often more information is required than the simply identity of system call. The exact type and amount of information varies according to the particular operating system and call. Two general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. However in some cases there may be more parameters than registers.

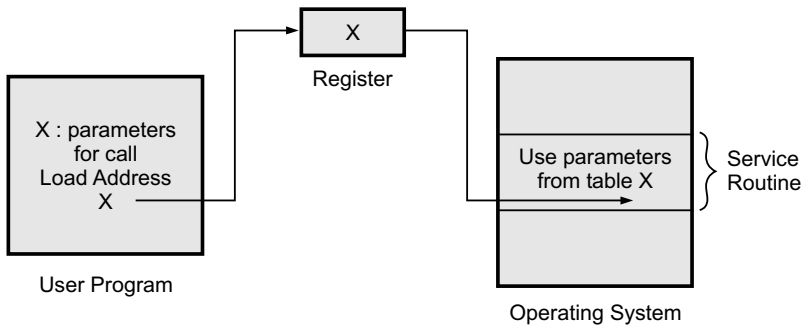


Fig. 5.9.9 Passing Parameters as a table

In these cases the parameters are generally stored in a block or table in memory and the address of block is passed as a parameter in a register as shown in Fig. 5.9.9.

5.9.10 How System Calls Used

Consider writing simple program to read data from one file and copy it to another. After obtaining the name of the two file, program must open the i/p file and create the o/p file. If in between some errors come, the program should print message on console (system call) and terminate abnormally (another system call).

When both the files are set up, we enter a loop which reads from the input file (a system call) and writes to the output file (another system call).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to console (more system calls) and finally terminate normally (the last system calls).

As we can see, programs may make heavy use of the operating system. All interaction between the program and its environment must occur as the result of requests from the program to the operating system. While programing in higher level languages programmer does not have to bother about internal implementations of system calls. The run-time support system for most programming languages provides a much simpler interface.

5.9.11 System Programs

Most of modern operating systems provide collection of system along with the services that it provides. System programs can be divided into several categories.

- File manipulation : These programs create, delete, copy rename, print, dump, list and generally manipulate files and directories.
- Status information : These program gives date, time, amount of memory or disk space, number of users, or similar status information.

File modification : These programs provides the facility of editing the files. It has several text-editors may be available to create and modify the contents of files stored on disk or tape.

Programming language support : Compile, Assemblers, and Interpreters for common programming languages such as Fortran, C, are often provided with the operating system.

Program loading and execution : The system may provide absolute loaders, relocatable loaders, leakage editors and overlay loaders.

Application program :

Along with operating system some particularly useful common programs such as compiler-compilers, text formatters, database systems and so on.

The most important system program for an operating system is its command interpreter. The command interpreter is that program which is running when a user first logs-in into a time-sharing system.

UNIX Operating System

UNIX is a powerful operating system. UNIX was developed by Bell Labs of AT & T.

5.9.12 Protocol Layers

The Table 5.9.1 shows the protocol layers of the open systems interconnection reference model.

Layer No.	Layer name	Main function	Example protocol	Network component
7	Application	Used for applications specifically written to run over the network.	FTP, DNS, NFS, http	gateway, smart switch
6	Presentation	Translates from application to network format, and vice versa.		gateway
5	Session	Establishers, maintains, and ends session across the network.	Named pipes, RPC	gateway
4	Transport	Additional connection below the session layer.	TCP	gateway
3	Network	Translates logical network address and names to their physical address (e.g., computer name to MAC address.)	IP	router, ATM switch
2	Data link	Turns packets into raw bits and at the receiving end turns bits into packets.	Ethernet	Bridge, network interface card
1	Physical	Transmits raw bit stream over physical cable.	IEEE 802	hub

Table 5.9.1 The OSI model layers

5.9.13 Features of Unix

1. It is a multiuser operating system.
2. It is a multitasking operating system.
3. It identifies each devices and directories as files.
4. It provides a high end security to files.
5. It provides communication and networking capabilities.
6. It provides portability i.e. it is easily runs on any platform or system.
7. It provides background processing, which allows a user to start new task but in background system continues its original task.
8. It provides UNIX Shell, shell is a powerful command interpreter.

5.9.14 Structure of Unix

As we explore the UNIX operating system, there are four main components : the Kernel, the shell, the file system and input/output.

5.9.14.1 The Kernel

The *Kernel* is the UNIX operating system. It is the master program that controls the computer's resources, allotting them to different users to different tasks. However, Kernel doesn't deal directly with a user, but it uses an interactive program called a shell. As UNIX is a multiuser system different users are demanding for resources the Kernel will decide which user should get the resource and when.

The shell acts as a interface between the user and system. When shell acts a "*command interpreter*", the shell takes your commands and handover this to Kernel, and Kernel will generate the system call and the command is executed.

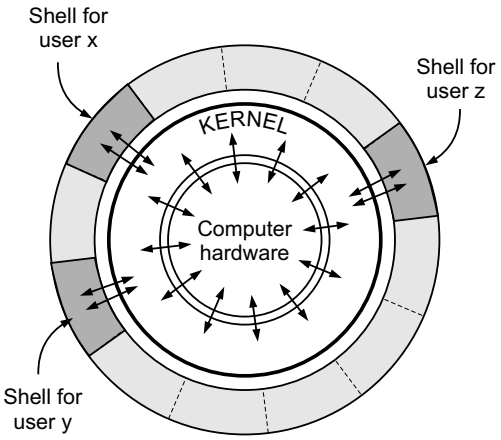


Fig. 5.9.10 The Kernel and the shells

When you login, the Kernel runs the *init* and *getty* system calls to check whether you are a authorized user and have the correct password. The Kernel also keeps track of all the programs being run, allotting time to each, also assigns storage for users files. Kernel runs shell programs and handles transfer of information between the computer and output devices. So we call the Kernel as Operating System. It's the heart of UNIX system, which is why it is called the Kernel.

Here are some Kernel duties in more details as shown in Fig. 5.9.10.

Time sharing and Processes

One of these most important Kernel duties is to run the time-sharing system. Multiuser means that more than one person can use the computer at the same time. Multitask means that even a single user can have the computer work on more than one task or program at the same time.

In UNIX each particular task or program that the computer undertaking is called a process. eg. When you run the date command that's one process. If you run it again, then that's a new process.

UNIX uses time-sharing to solve the problem which arises because of multiple demands. Time sharing means the Kernel maintains a list of current tasks (or processes)

and allots a short span of time to one process, then to the next and so on. As Kernel allots a short span of time to each processes some times it may not be possible to finish that process in a single cycle, it may requires several cycles of time sharing.

At any one time on a UNIX system, several processes are active. The one process that is actually running at a given movement is said to have run status. The others that are waiting for their turn are usually have sleeping status.

If user wants to known which are the processes running currently he can use the command ps (process status) for this.

5.9.14.2 The ps Command

If you type *ps* you will get output that looks something like this :

PID	TTY	TIME	COMMAND
2345	12	0:12	sh
2387	12	0:02	ps

The ps command makes a process status report. When a process is initiated the kernel assigns to it a unique *PID* (process identification number). In our example the only two processes are the user's shell itself (sh) and the ps command. Each has its own PID. Looking at the PIDs, we see that some 41 other process were created between these two, probably by other users, your shell which you are using is also a another process for Kernel.

The sh process in above example is called the parent, and the ps process spawned by the shell is the children. The parent is said to wait until the child dies before resuming running status.

5.9.14.3 The Kernel and System Calls

As Kernel acts as an interface between the shell and the UNIX commands on the one hand and the system hardware on the other. The Kernel is the part of UNIX that directly communicates with the hardware.

We don't need to know what goes inside the machine in order to execute UNIX commands. A user can also write a new program without knowing innermost secretes of hardware, using system calls.

In UNIX, system calls are basic routines used in the Kernel for doing specific basic tasks. If you use a system call to open a file, then the part of the Kernel that opens files is invoked in your behalf. Each system call accomplishes a specific basic task. One system call opens file, another closes file and so on.

These systems calls can be used in programming for a command directly or indirectly, by using UNIX library functions that internally uses system calls.

5.9.14.4 Running A Command : The Shell

Shell, it is the actual interface between user and the computer what happens when you type a command such as date? Answer is that the current time and date appear on your terminal screen, but what goes on behind the screen.

When a user log in the Kernel starts up a program, the shell that handles users interaction with system. The shell sends a prompt symbol typically a $ for Bell release to the screen, then waits for input from user. When user type the command date and hit return key. The shell obtains the users command, executes it if possible, and returns prompt to user when it is done and ready for next command. The Kernel does the actual fetching and returning of data at the request of the shell. As shell translates users typed commands to actions it is termed as Command Interpreter. When a command date is given, the shell searches for a file name date. If it finds such a file and if the file contains an executable program, the shell starts the program running and goes into a waiting mode. When the program finishes, the shell resumes. For command file, shell searches through a predefined list of directories given by the PATH shell variable. Typically this list includes your current working directory and system directories devoted to storing files of commands.

To see what directories are in your search path, type

```
echo $ PATH
```

The response will be something like this :

```
. :/bin:/user/bin
```

There the colon is used to separate directories, and the period stands for your current working directory. This example instructs the shell to look for a command first in your current directory, then in the /bin directory and finally in the /user/bin directory. If it doesn't find the command in one of the directories, the shell gives the message command was not found.

5.9.14.5 A Program Hierarchy

The UNIX system includes a large number of programs that user can use. These programs are arranged in a hierarchy of dependence. The most basic programs are the system calls, which are part of Kernel program. Next there is library of programs in C programming language. These C language programs use the system calls to handle the interface between the programs and hardware. The programs written using these system calls are *portable*, means these program user can transfer from one UNIX machine to another and execute it.

Then there are the UNIX commands. The commands are programs like who, cat and ls and so on. These commands are written in C language and system calls and /or C library programs.

Finally, some commands are shell scripts. That is, they are files that contain a list of other UNIX commands. When the shell script command is run, what actually happens is that the list of commands in the file gets run.

5.9.15 Files and Directories

As shell, Kernel, system calls, commands - the code for all these programs is stored in files on a UNIX system. So we will see how is file system in UNIX. UNIX has got a tree structured UNIX file and directory system. Fig. 5.9.11 shows the UNIX directory structure.

In UNIX every file is assigned to a directory. A directory is a special form of file that maintains a list of all the files in it. As UNIX identifies each device, directories as a file so, any file may be directory itself, it would be called a subdirectory of the original. This makes the UNIX file structure like the tree of directories and files. You can visualize the directories as being the various limbs, branches and the tree, while the files as the leaves attached to them. Each user has a home directory and is free to create her own subdirectories, sub-subdirectories, and so on. The commands ls, rm are breaking down if subdirectories are more than 35.

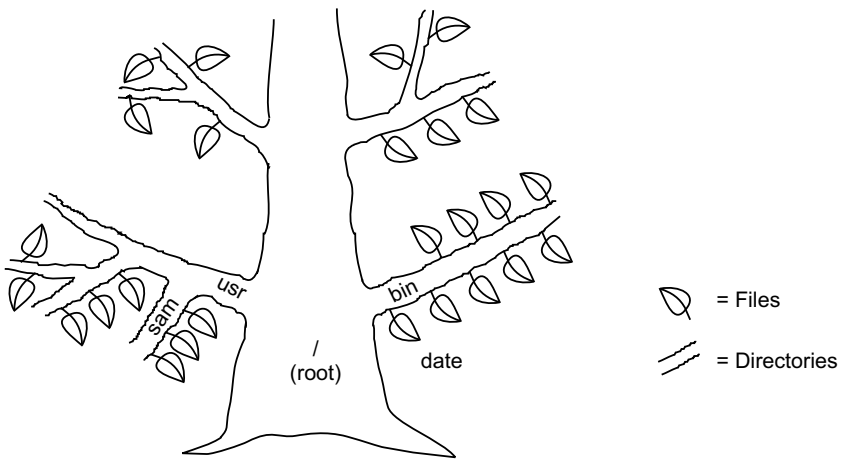


Fig. 5.9.11 The UNIX directory system

The root directory, which is named /, is the prime directory. All other directories are subdirectories. The tree scheme makes it easy to organize files. The UNIX *cd* command can be used to change from one directory to another. pwd UNIX command prints out the name of users current working directory.

The full pathname for file is like /usr/nemo/pet/fido. Here, the first-slash(/) stands for root. The other slashes used to separate each directory name from the next.

5.9.15.1 File Creation

If user needs to create a file using a editor, the several things happens in UNIX.

First, the Kernel allots space in storage disk to hold the file. The file hold the text containing digits, alphabets and some other characters.

Second, an inode entry is created on a section of the disk. The inode contains nearly all there is to know about the file. This information includes the location on the disk, where file starts, the size of file, when the file was past used, the various read, write and execute permissions are who owns the file and other administrative information. Each file has unique inode number, and by this number system refers to the file. Only the file name file contents are not in inode.

Third, a directory entry is created. A directory is just a file itself, and each directory entry consist of just two items : the inode number of a file and the user selected name of that file as shown in Fig. 5.9.12.

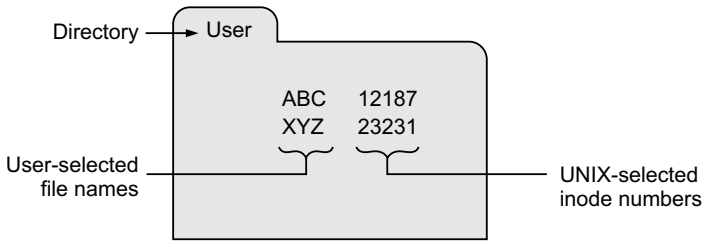


Fig. 5.9.12 Directory entries for the user directory

The contents of a file are stored at one place, most of relevant information in a second location and name in third as shown in Fig. 5.9.12.

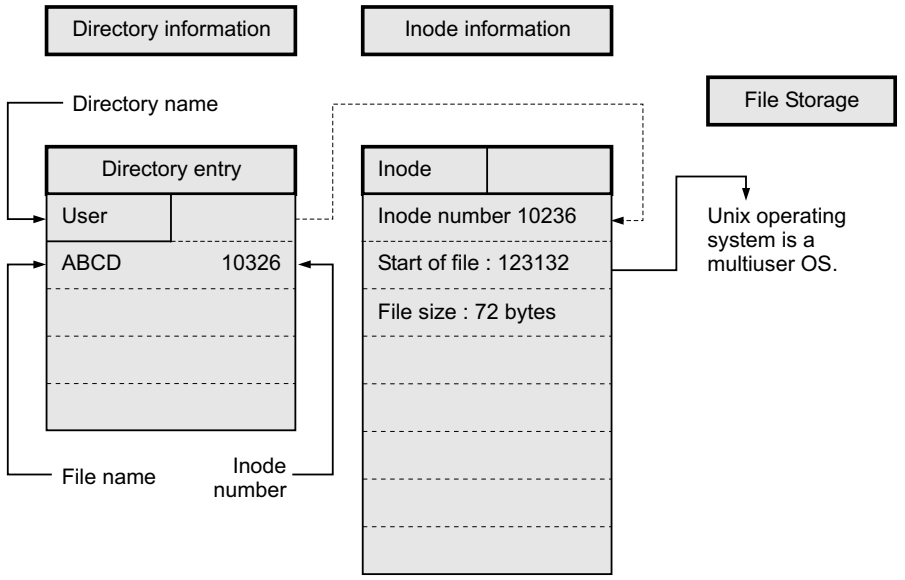


Fig. 5.9.13 Directory entry, entry and file contents

5.9.15.2 File Systems

What does mean *file system*? Answer to this is how files and inodes are stored on a system. The disk or portion of a disk is set aside to store files and inode entries. The entire functional unit is termed on a file system. This disk memory is divided into blocks typically 512 bytes in size. These blocks are organized into three groupings : The first group consist of block 1 and is called the *super block*. It contains information about that particular file system. Fig. 5.9.14 shows file system for example it tells how many blocks are in the entire system, how many blocks devoted to inodes, and how many blocks are free. The second group consist of the blocks devoted to inodes, and third and final portion of the file system consist of blocks for storing the files themselves.

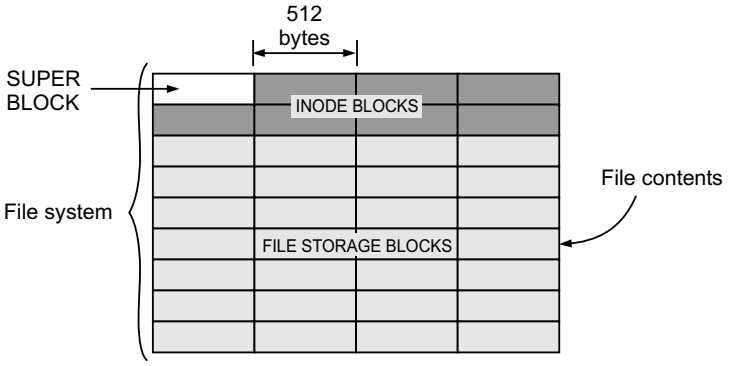
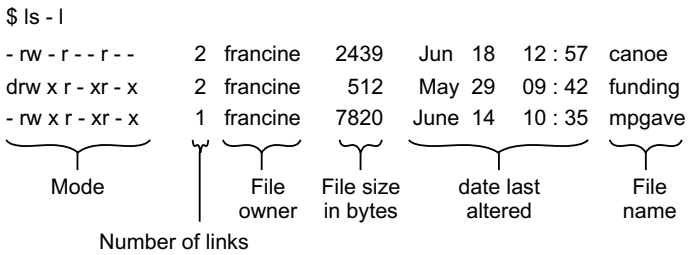


Fig. 5.9.14 A file system

5.9.16 Peripheral Devices : Special Files

Each device connected to a UNIX system is represented by a file in the /dev directory. The *ls* command is used to list the files in a directory. If we used the command ls-l on a regular directory containing a subdirectory and some ordinary files we get the result as shown below.


$ ls - l
- rw - r - - r - - 2 francine 2439 Jun 18 12 : 57 canoe
drw x r - xr - x 2 francine 512 May 29 09 : 42 funding
- rw x r - xr - x 1 francine 7820 June 14 10 : 35 mpgave
Mode
Number of links
File owner
File size in bytes
date last altered
File name

The output begins with a 10-character string of dashes and letters called the mode as shown in Fig. 5.9.15. If the first character is a "d", then the file is a directory. If the first character is a"-", then the file is an ordinary file. The next 9 characters in the string describe the file permission. The first set of three characters, refer to user, the second set to other members of user's group, and the third group is everyone else. An "r" in the first position within each set indicates permission to read the file, a "w" in the second

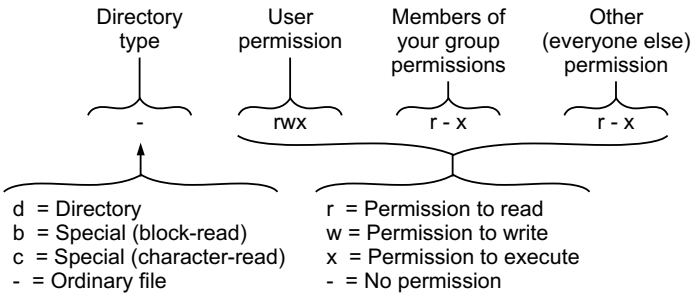


Fig. 5.9.15 The mode

position within each set indicates permission to write in the files and an "x" in the third position indicates permission to execute. A"-" in any of these locations indicates no permission.

If we apply the same command to the /dev directory ; the result will be shown in the Fig. 5.9.16.

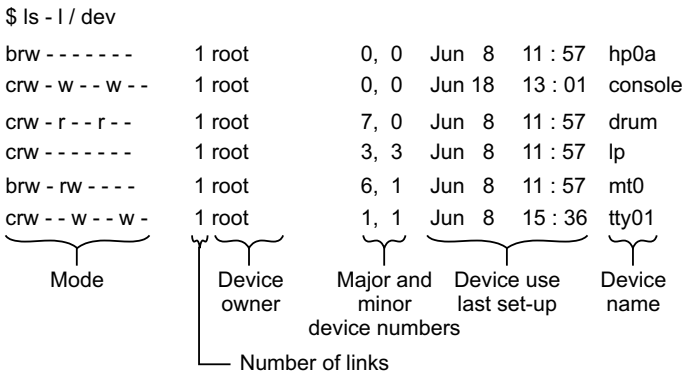

$ ls - l / dev
brw - - - - - - - 1 root 0, 0 Jun 8 11 : 57 hp0a
crw - w - - w - - 1 root 0, 0 Jun 18 13 : 01 console
crw - r - - r - - 1 root 7, 0 Jun 8 11 : 57 drum
crw - - - - - - - 1 root 3, 3 Jun 8 11 : 57 lp
brw - rw - - - - 1 root 6, 1 Jun 8 11 : 57 mt0
crw - - w - - w - 1 root 1, 1 Jun 8 15 : 36 tty01
Mode
Device owner
Number of links
Major and minor device numbers
Device use last set-up
Device name

Fig. 5.9.16 - ls - l/dev

The devices listed are the terminals they are tty01, tty02 of and so on. The hp entries are disk drives, mt indicates a tape drive and lp is a printer.

The ls - l output for /dev is slightly different from the one for regular files.

First the first character of the mode string is a c or a b rather than a - (hyphen) or a d. These label the devices as character-devices (c), like the terminals, and block-devices (b), like the tape drives. The distinction is between devices that transmit data character by character (c) and those that use blocks of data (b). The presence of either of these labels tells UNIX that the file is a "special file", one associated with device.

The second new point is that the size parameter in the listing is replaced by a pair of numbers separated by comma. The major number identifies the type of device, and the minor number labels the individual specimens of that type.

This new information (file type and device numbers) is stored in the inodes for these special files. Thus when kernel inspects the inode, it learns that it is dealing with a device and what kind of device it is.

5.10 Multiprocessors

May-11

If a microprocessor system contains two or more components that can execute instructions independently then the system is called multiprocessor system. Its main aim is to improve throughput, reliability, flexibility and availability. The Fig. 5.10.1 shows typical multiprocessor organization. It consists of two or more processors of approximately comparable capabilities. All processor share access to common sets of memory modules, input-output channels, and peripheral devices. Along with shared memory and input-output devices, each processor has its own local memory and input-output devices. The interprocessor communication is done through the shared memories or through an interrupt network. Most importantly, the entire system is controlled by the single operating system providing interactions between processors and their programs at various levels.

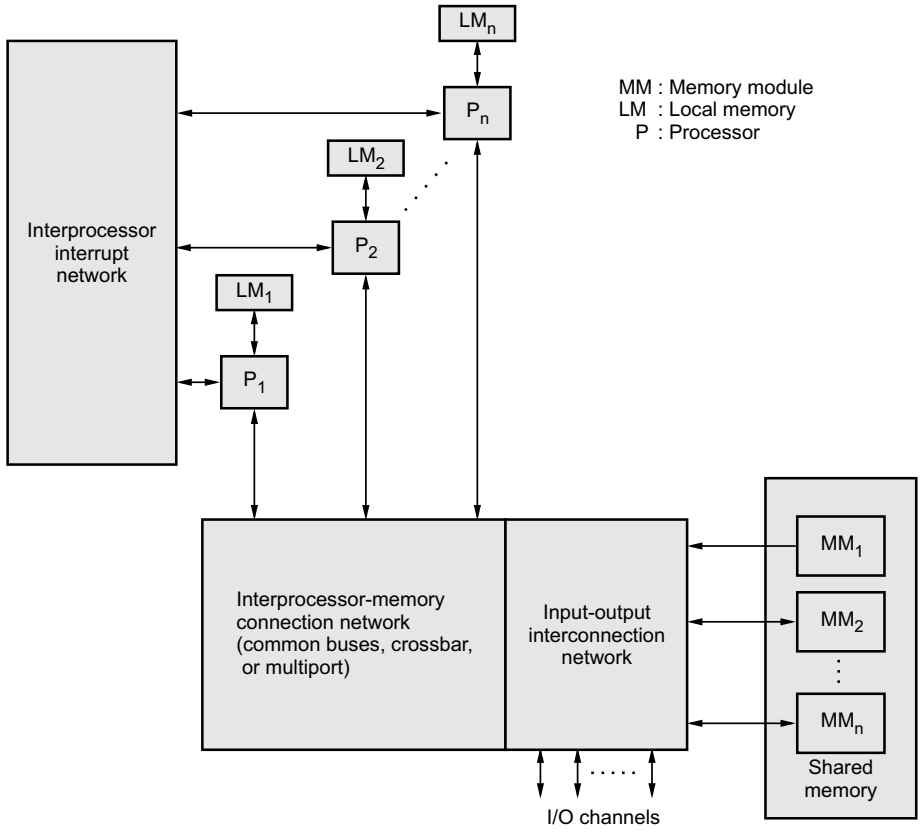
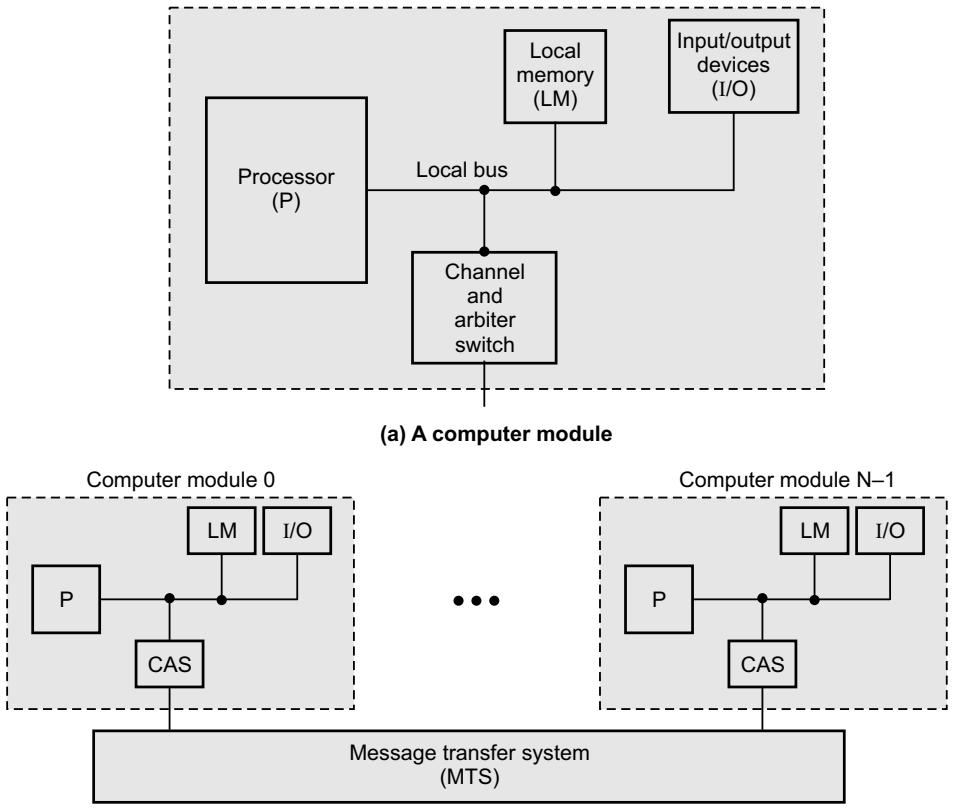


Fig. 5.10.1 Function design of typical multiprocessor system

The multiprocessor systems are implemented using one of the two basic architectures : loosely coupled architecture and tightly coupled architecture. The systems using these architectures are known as loosely coupled systems and tightly coupled systems respectively.

5.10.1 Loosely Coupled Multiprocessors (LCS)

The Fig. 5.10.2 shows nonhierarchical loosely coupled multiprocessor system. In loosely coupled systems, each processor has a set of input-output devices and a large local memory where it accesses most of the instructions and data. The processor, its local memory and input-output interfaces are together called **computer module**. Processes which execute on different computer modules communicate by exchanging messages through a **Message Transfer System (MTS)**. The coupling in such a system is very loose. Hence, such systems are also referred to as a **distributed systems**. These systems are usually efficient when the interactions between tasks are minimal.



(a) A computer module

(b) Loose coupling of computer modules

Fig. 5.10.2 Nonhierarchical loosely coupled multiprocessor system

As shown in Fig. 5.10.2, a computer module of a nonhierarchical loosely coupled multiprocessor system. It consists of various computer modules connected to the message transfer system. Each computer module consists of a processor, a local memory, local input-output devices and an interface to other computer modules. The interface to the other module usually consists of channel and arbiter switch (CAS).

The computer modules initiate communication with other modules by requesting MTS for the service. In case of simultaneous requests, the arbiter is responsible for choosing one of the simultaneous requests according to a given service discipline. The arbiter is also responsible for delaying other requests until the servicing of the selected request is completed. The channel within the CAS usually consists of a high speed communication memory which is used for buffering block transfers of messages. It is accessible by all processors.

The message-transfer systems for a nonhierarchical LCS are divided into two categories : 1. Simple time shared bus or 2. Shared memory system.

The time shared bus is a common communication path connecting all functional units. On the other hand shared memory system consists of a set of memory modules and a processor-memory interconnection network or a multiported main memory. In a multiported memory system, the arbitration and selection logic for the access control are distributed into the memory modules. The configuration of MTS is one of the most important factors that determine the performance of the multiprocessor system. In LCS configurations with a single time shared bus, the performance is limited by message arrival rate on the bus, the message length and bus capacity (in bits/sec). In LCS that use a shared memory MTS, the limiting factor is the memory conflict problem due to the processor memory interconnection network.

Processes (tasks) can communicate with other processes allocated to the same processor, or with tasks allocated to other processors. With each task is an input port stored in the local memory of the processor to which the task is allocated. The message is directed to the input port of the destination task. Communication between tasks allocated to the same processor take place through the local memory. However, communication between tasks allocated to different processors is through a communication port with in the communication memory. The processor is allotted with one communication port as its input port.

5.10.2 Tightly Coupled Multiprocessors

Tightly coupled systems (TCS) communicate through a shared main memory. Hence the rate at which data can communicate from one processor to the other is on the order of the bandwidth of the memory. One of the limiting factors to the expansion of a TCS is the performance degradation due to memory contentions which occur when two or

more processors attempt to access the same memory unit concurrently. However, when high-speed or real-time processing is desired, tightly coupled systems (TCS) may be used. There are two models of a TCS :

1. TCS without private cache 2. TCS with private cache.

5.10.2.1 TCS without Private Cache

The Fig. 5.10.3 shows the tightly coupled multiprocessor system without private cache. It consists of P processors, M memory modules and C input-output channels. These units are connected through a set of three interconnection networks, viz.

1. The processor-memory interconnection network (PMIN)
2. The input-output processor interconnection network (IOPIN)
3. The interrupt-signal interconnection network (ISIN)

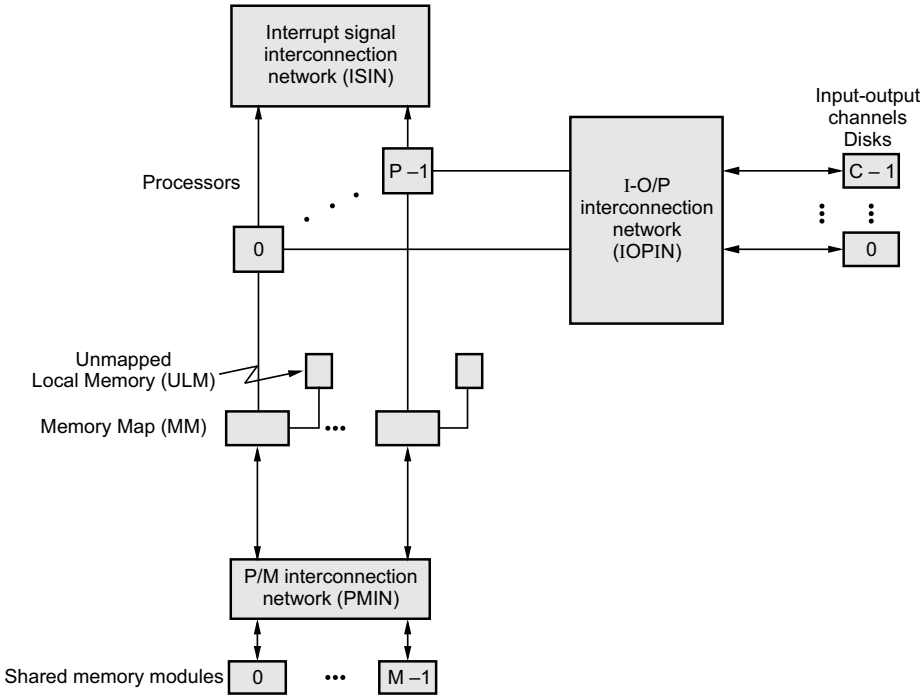


Fig. 5.10.3 TCS without private cache

The PMIN is a switch which is used to connect every processor to every memory module. This switch is P by M crossbar with PM sets of cross points. A set of cross points for a particular processor-memory pair includes (n + k) cross points, where n is the width of address within a module and k is the width of the data path. Hence the crossbar switch for a P by M multiprocessor system has a complexity O(PM (n + k)). When the crossbar switch is distributed across the memory modules, the system is known as a multiported memory system.

A memory can satisfy only one processor's request in a given memory cycle. Hence, if two or more processors attempt to access the same memory module, a conflict occurs which is resolved or arbitrated by PMIN. To avoid excessive conflicts the number of memory module L is usually as large as P. Another method to minimize conflicts is to associate a reserved storage area with each processor. This is the unmapped Local memory (ULM). ULM is used to store Kernal code and operating system tables often used by the processes running on that processor. The IOPIN is used to allow a processor to communicate with an I/O channel which is connected to peripheral devices. The ISIN is used for two purposes : To direct an interrupt to any other interprocessor network and to initiate hardware alarm in case of processor failure.

5.10.2.2 TCS with Private Cache

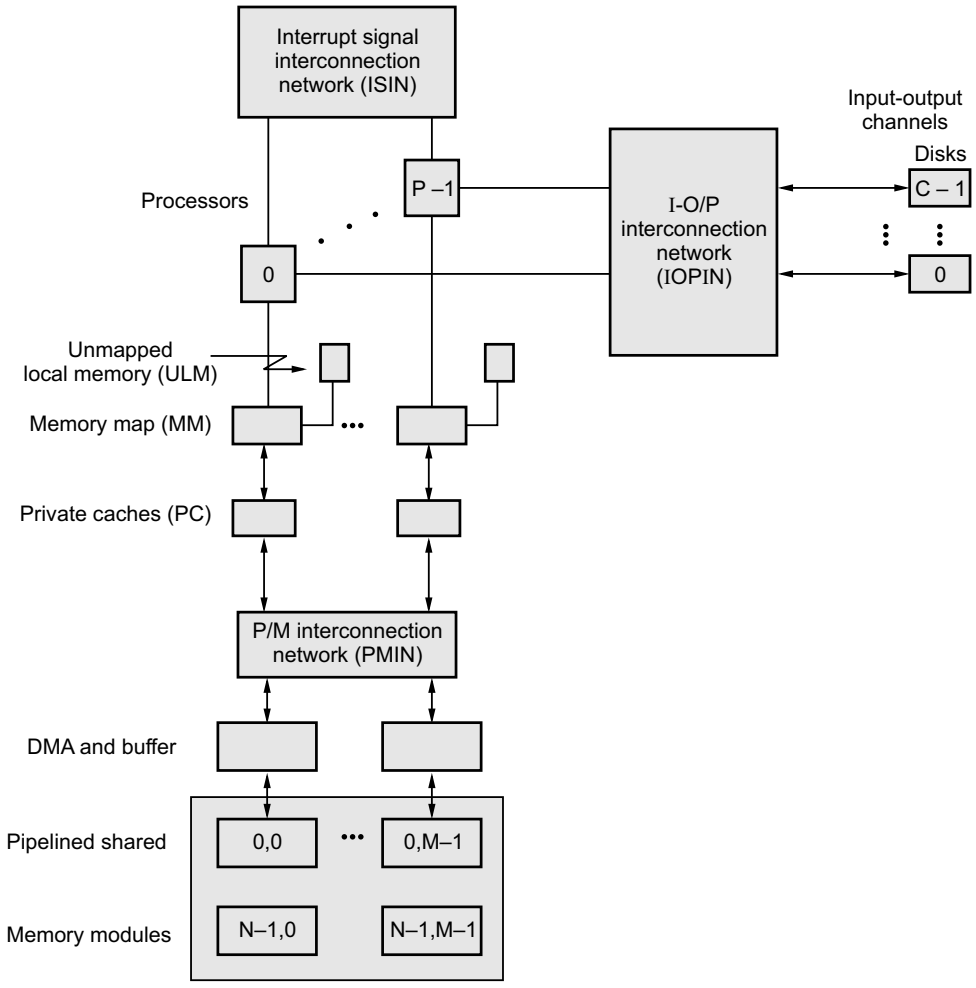


Fig. 5.10.4 TCS with private cache

In the first model (that is without private cache) each memory reference goes through the PMIN, it encounters delay in the process or memory switch and hence the instruction cycle time increases. The increase in the instruction cycle time reduces the system throughput. This delay can be reduced by associating a cache with each processor to capture most of the references made by a processor. Another advantage of the cache is that the traffic through the crossbar switch can be reduced, which subsequently reduces the contention at the cross points. More than one inconsistent copy of data may exist in the system as this multiprocessor organization encounters the cache coherence problem.

5.10.3 Processor Characteristics for Multiprocessing

The desirable architectural features for a processor to be effective in a multiprocessing environment are :

1. Process recoverability.
2. Efficient context switch.
3. Large virtual and physical address space.
4. Effective synchronization primitives.
5. Inter processor communication mechanism.
6. Instruction set.

1. Process recoverability

- The architecture of a processor used in multiprocessor system should consider that the process and processor are two different entities.
- It processor fails, then another processor should retrieve the interrupted process state so that execution of the process can continue.
- Most processors contain the process state of current running process in internal registers which are not accessible from outside the processor. Therefore, the processor should have separate the general purpose registers to access other processor state in case of failure without much loss of speed.
- The register file should be shared by all processors in case of some operation modes.

2. Efficient context switching

- The processor should support efficient context switching operation. It is necessary for the processor to support more than one addressing domain. The context switch operation saves the state of current process, restores the state of new process and switches to a selected ready-to-run process.

- The architecture should support a special instruction to accomplish the context switch efficiently. For example, **Central Exchange Jump** (CEJ) provided in the cyber 170 processor which has a single set of registers. The execution of **CEJ** results in saving of the context or state of the current processor and the register set replaced by the state of another process. Such save and restore operations for all processor registers can significantly contribute to the overhead in achieving concurrency in the system.
- As shown in Fig. 5.10.5, architecture should provide ample number of register sets to accomplished task switching efficiently by changing the contents of **current process register** in the processor to point the register set containing the state of selected process. For example, Each processor in S-1 multiprocessor system has 16 sets of registers.

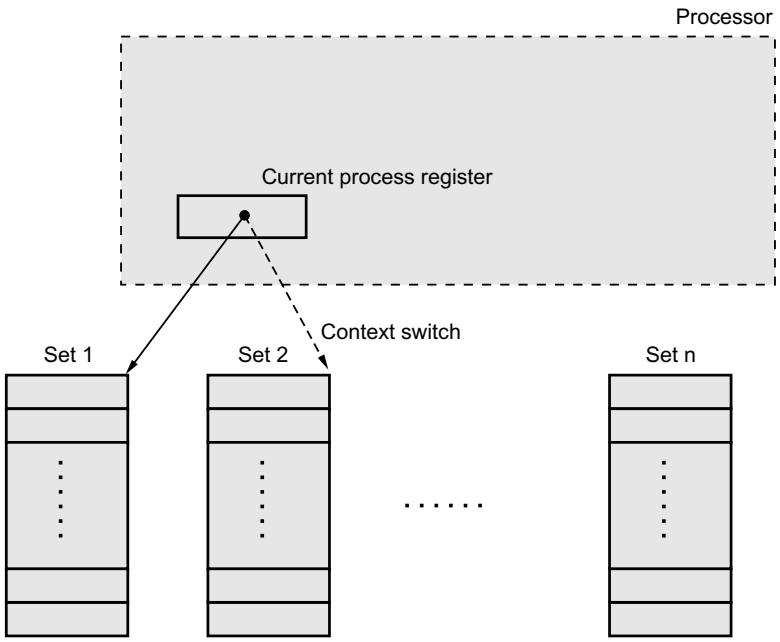


Fig. 5.10.5 Multiple register sets for efficient context switching

3. Large virtual and physical address space

- Processor sometimes needs to access large amounts of data objects. To satisfy this requirement it should support large physical address space.
- It should also provide a large virtual address space. If possible, the virtual address space should be segmented to achieve modular sharing and provide mechanism for memory protection and software reliability. For example, S1-multiprocessor system's processor has 2 gigabytes of virtual memory and 4 gigawords of physical memory, where each word is 36 bits wide.

4. Effective synchronization primitives

- The processor design must provide some mechanism for the synchronisation of indivisible actions to provide mutual exclusion. Mutual exclusion is required when two or more processes are in execution concurrently and must cooperate to exchange data.
- Mechanisms to achieve mutual exclusion consists of some form of read-modify-write memory cycle and queuing. One such mechanism is **semaphore**.
- Each semaphore has its own queue and the entries in the queue refer to processes which are suspended because of the semaphore value.
- The queue manipulations should be efficient.
- Instructions like test-and-set, compare-and-swap should be used to achieve mutual exclusion.

5. Interprocessor communication mechanism

- The interprocessor communication mechanism should be provided and implemented in hardware as it becomes efficient and easy when there are frequent requests for services exchanged between different processors with synchronisation.
- If a processor fails, then the hardware mechanism should send signal to other processors for error recovery or diagnostic procedure.
- The processors must have a wait state or some mechanism to suspend the processor in a queue. So that each processor can participate in the arbitration of the requests.

6. Instruction set

The instruction set of the processor in the multiprocessor system should facilitate :

1. Implementation of high-level languages
2. Procedure linkage
3. Looping constructs
4. Parameter manipulation
5. Multi dimensional index computation
6. Range checking of addresses
7. Instructions for creating and terminating parallel execution paths within a program
8. Hardware counters and read time clocks to generate a unique name of process identification and time out signals required for process management.

5.10.4 Interconnection Networks

The important characteristic of a processor used in multiprocessor system is its ability to share a set of main memory modules and possibly I/O devices. This sharing capability is provided through a set of two interconnection networks.

1. Between processor and memory modules.
2. Between processor and I/O subsystem.

There are several different physical forms available for the interconnection networks. Let us study them.

5.10.4.1 Time Shared Bus or Common Bus

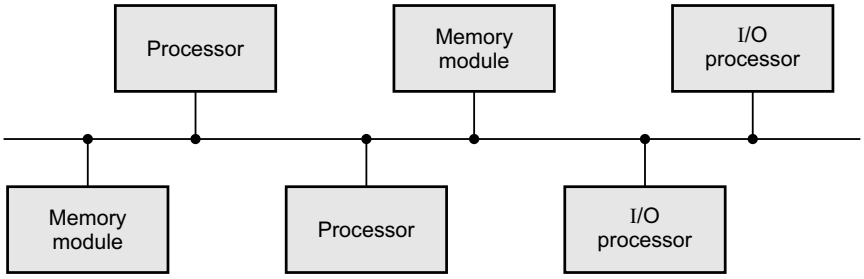


Fig. 5.10.6 Single-bus multiprocessor organization

In a time shared bus interconnection system for multiple processors provides a common communication path connecting all of the functional units. The Fig. 5.10.6 shows single bus multiprocessor organization with a common communication path. The organization shown in Fig. 5.10.6 is simplest and easiest to reconfigure. Such interconnection networks do not have any active components; they are totally passive. In these networks, transfer operations are controlled completely by the bus interfaces of the sending and receiving units.

The conflict-resolution methods such as static or fixed priorities, first-in-out (FIFO) queues and daisy chains are used to decide the access to the common bus. The example of PDP-11 Unibus. It has 56 signal lines to provide the control, address lines and data paths necessary to transfer 16-bit word.

The advantage of single-bus organization is that it is quite reliable and relatively inexpensive. However, it can cause complete system failure as a result of malfunctioning in any of the bus interface circuits.

Such organization is that as number of processors and memory increases, the chances of the bus contention problem also increases. This decreases system throughput and increases arbitration logic. To solved some of the problems mentioned above without an appreciable increase in system complexity or decrease in reliability usually two

unidirectional buses are used, as shown in Fig. 5.10.7. However, a single transfer operation in such a system usually requires the use of both buses.

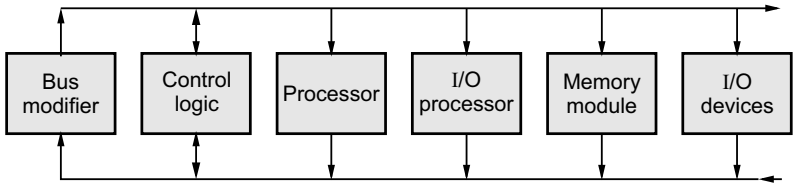


Fig. 5.10.7 Multiprocessor with unidirectional buses

The increase in system complexity and decrease in reliability due to two buses suggested to use multibus bidirectional buses as shown in Fig. 5.10.8.

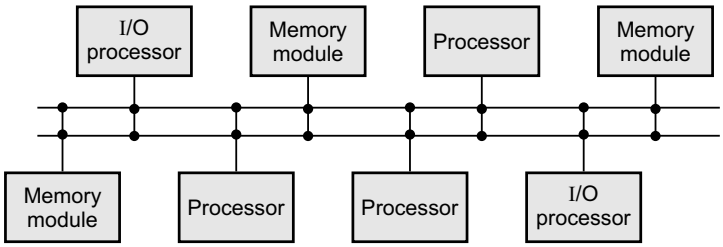


Fig. 5.10.8 Multi-bus multiprocessor organization

Multibus permits multiple simultaneous bus transfers but increases in system complexity. Alongwith the organization there are several factors that affect the characteristics and performance of a bus. These factors are :

- Number of active devices on the bus
- The bus-arbitration algorithm
- Centralization or distribution of control
- Data width
- Synchronization of data transmission and
- Error detection

5.10.4.2 Cross Bar Switch

If the number of buses in a common bus system is increased, a point is reached at which there is a separate path available for each memory module. Fig. 5.10.9 shows the crossbar switch system organisation for multiprocessors which provides separate path for each memory module. The interconnection network shown in Fig. 5.10.9 is called **nonblocking crossbar**.

The crossbar switch has a separate bus associated with each memory module. Therefore, the maximum number of transfers that can take place simultaneously is limited by the number of memory modules and the bandwidth-speed product of the buses rather than the number of paths available.

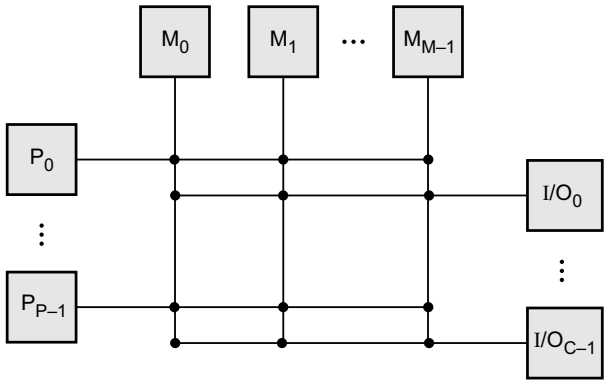


Fig. 5.10.9 Crossbar switch system organization for multiprocessors

Important characteristics of a system utilizing a crossbar matrix are :

1. Extreme simplicity of the switch-to-functional unit interfaces.
2. Ability to support simultaneous transfers for all memory units.

To provide these features, it requires hardware capabilities which are capable of switching parallel transmissions and capable of resolving multiple requests for access to the same memory module occurring during a single memory cycle.

Functional Structure of a Crosspoint in a Crossbar Network

The Fig. 5.10.10 shows the functional structure of a crosspoint in a crossbar network. It consists of multiplexer, memory and arbitration modules. Each processor activates a memory module request signal (REQ) to arbitration unit. The arbitration module selects the processor with highest priority using a priority encoder. After selecting a processor,

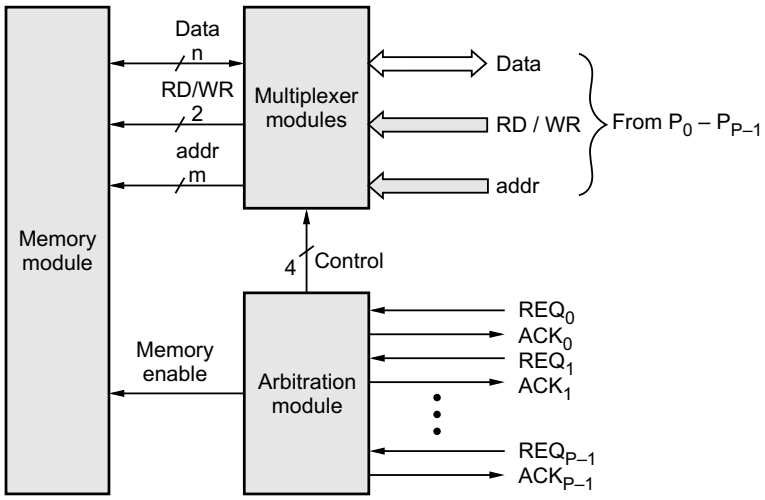


Fig. 5.10.10 Functional structure of a crosspoint in a crossbar network

arbitration module return an acknowledge signal (ACK) to it. On receiving the ACK signal, processor initiates the memory operation. Multiplexer module is used to connect data, control and address signals from the selected processor to the memory module.

The required flexibility in access to the input/output devices can be provided by using a switch on the device side of the input/output processor or channel.

5.10.4.3 Multiport Memory

In some systems the control, switching, and priority arbitration logic are distributed throughout the crossbar switch matrix which is distributed at the interfaces to the memory modules. Such a system is called **multiport memory system**. Such system configuration is well suited to both uni and multiprocessor organizations. Here, memory-access conflicts are resolved by assigning permanently designated priorities at each memory port.

In these organizations, it is possible to designate portions of memory as private to certain processors, input/output units or combinations thereof, as shown in Fig. 5.10.11.

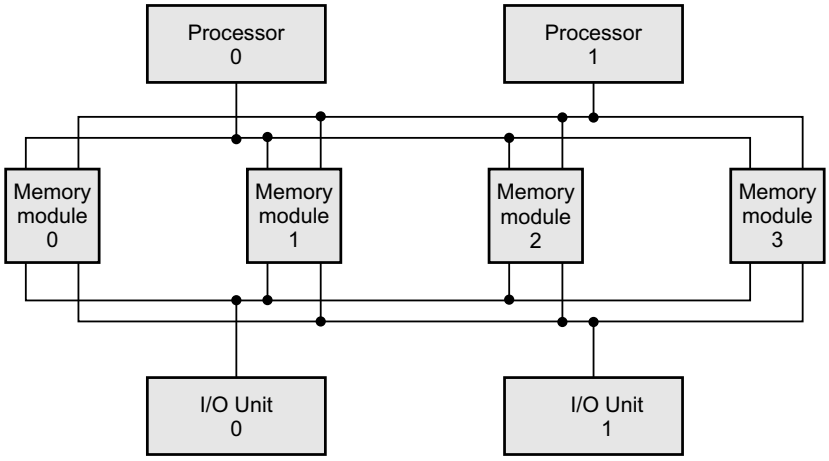


Fig. 5.10.11 Multiport memory without fixed priority assignment

In Fig. 5.10.12 memory modules 0 and 3 are private to processors 0 and 1 respectively. Such organisation has an advantage that due this there is increase in protection against unauthorized access and it also permit the storage of recovery routines in memory areas that are not susceptible to modifications by other processors.

However, the main disadvantage of such system is the system recovery if the other processors are not able to access control and status information in a memory block associated with a faulty processor. This organization can also support non blocking access to memory if a full-connected topology is used.

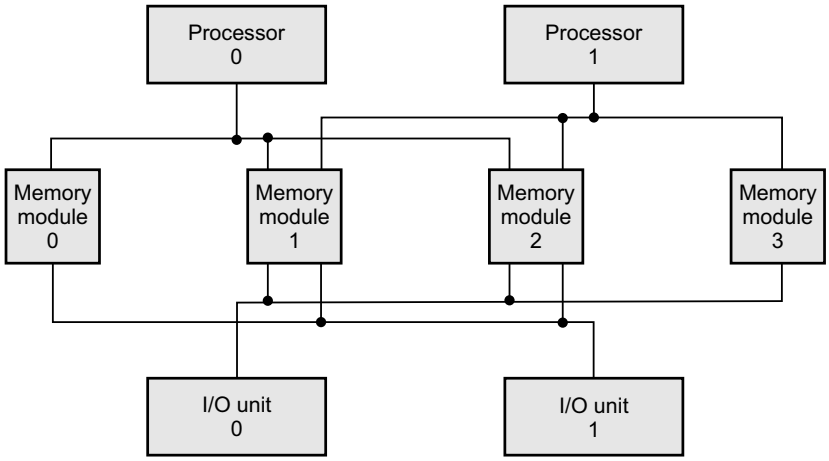


Fig. 5.10.12 Multiport memory with private memories

Important points related to three multiprocessor hardware organizations

Sr. No.	Time shared bus	Crossbar switch	Multiport memory
1.	Lowest overall system cost for hardware and least complex.	Most complex. Highest total transfer rate.	Most expensive since most of control and switching circuitry is in the memory unit.
2.	Easy to modify hardware system configuration.	Functional units are simplest and cheapest.	Functional units permit low cost uniprocessor.
3.	Overall system capacity limited by bus transfer rate. Bus fail then whole system fail.	Usually cost-effective for multiprocessors only as basic switching matrix is required to assemble any functional units into a working configuration.	Potential for a very high total transfer rate in overall system.
4.	Expanding may degrade system performance.	System expansion improves overall performance.	Difficult to modify as the design decision is made quite early.
5.	Lowest efficiency.	Expansion of system is limited only by size of the switch matrix.	Large number of cables and connector are required.
6.	Organization is suitable for smaller systems only.	Reliability of system can be improved by segmentation or redundancy within switch.	

Table 5.10.1

5.10.4.4 Multistage Interconnection Network (MIN)

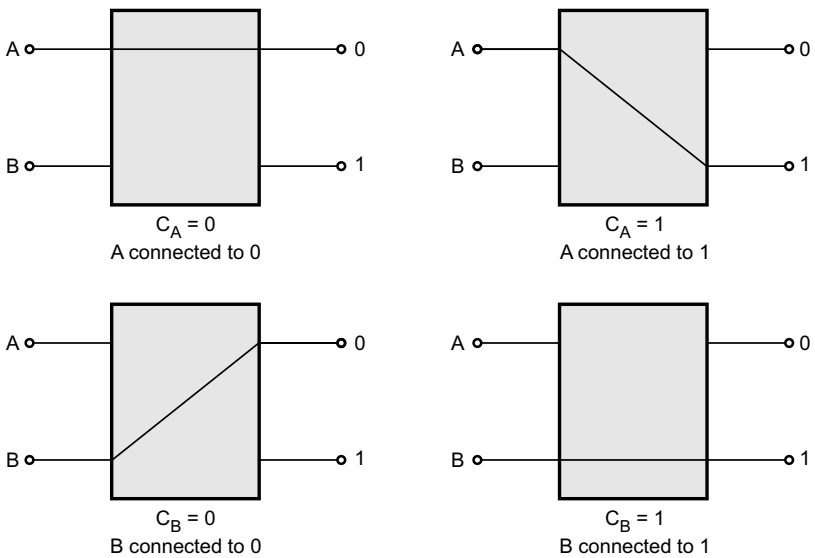
The important characteristic of a processor used in multiprocessor system is its ability to share a set of main memory modules and possibly I/O devices. This sharing capability is provided through a set of two interconnection networks. These are

- Time shared bus or common bus
- Cross bar switch
- Multiport memory
- Multistage switching network
- Hypercube system

We have already seen first three interconnection networks. Let us see last two interconnection networks.

Multistage Switching Networks

In multistage switching networks a 2 × 2 crossbar switch is used as a basic component. It has two inputs, labeled A and B, and two outputs, labeled 0 and 1. The control inputs C_A and C_B associated with the switch establish the connection between the input and output terminals as shown in the Fig. 5.10.13. If control input is zero it connects the input to the 0 output and if it is one it connects the input to the 1 output. The 2 × 2 switch also has the capability to arbitrate between conflicting requests. If both inputs A and B require the same output terminal, then only one of them will be connected and the other will be blocked or rejected.



5.10.13 2× 2 crossbar switch

Using the 2 × 2 crossbar switch as a building block, it is straightforward to construct a multistage network to control the communication between a number of sources and destinations as shown in the Fig. 5.10.14. The connections are accomplished by constructing a binary tree of cross bar switches. With this connection it is possible to connect the input to one of eight possible destinations. Consider that two inputs are the two processors P_A and P_B and they are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to destination goes through three levels. One binary bit is assigned to each level to choose output in that level. The first bit of the 3-bit destination number determines the switch output in the first level, the second bit specifies the output of the switch in the second level, and the third bit specifies the output of switch in the third level. For example, to connect P_B to memory module 011, it is necessary to form a path from P_B to output 0 in first level, output 1 in the second level and output 1 in the third level.

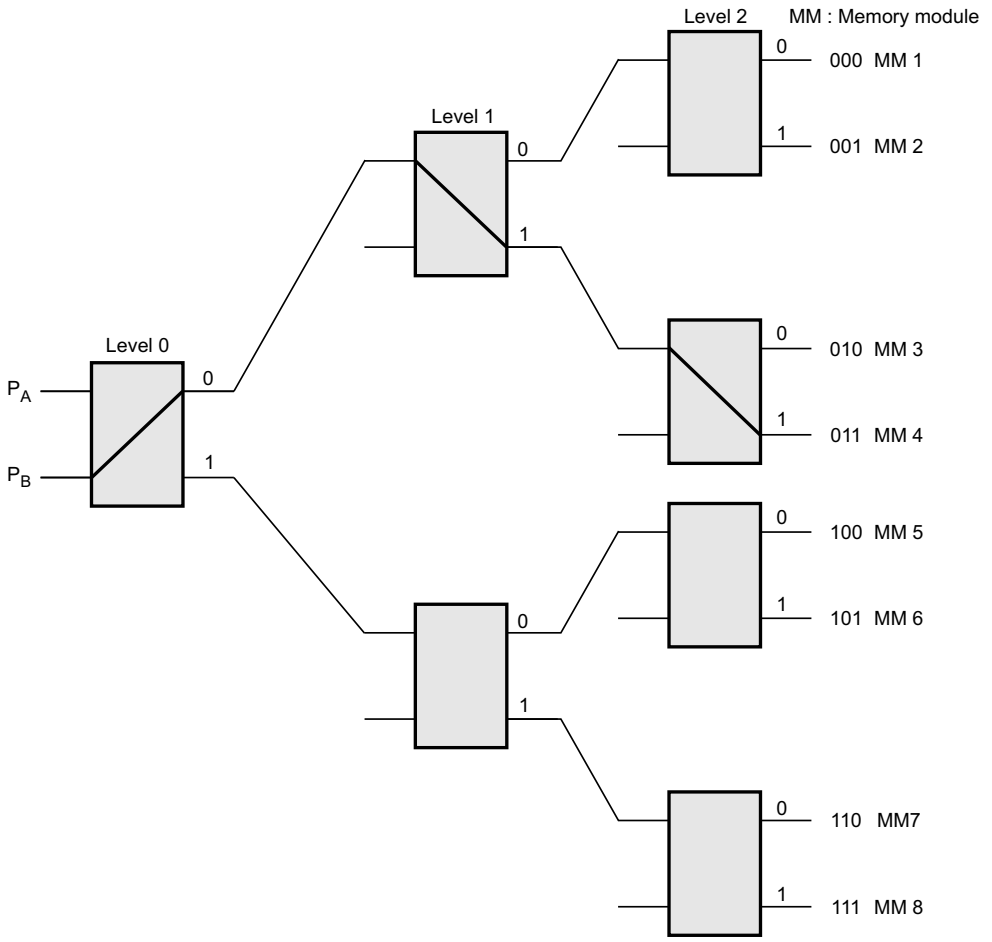


Fig. 5.10.14 1 to 8 way switch using basic 2 × 2 switch

Many different patterns of 2 × 2 switches can be constructed to form different multistage networks such as omega network, butterfly network and so on. Let us see the construction of omega network, as shown in Fig. 5.10.15. As shown in the Fig. 5.10.15, four possible connections of 2 × 2 switches used in constructing the 8 × 8 omega network. Three levels of 2 × 2 switches are needed to construct 8 × 8 network. In general, an n-input omega network has $\log_2^n$ levels. The levels are labeled from 0 to $\log_2^{n-1}$ from the input end to the output end. Data routing is controlled by inspecting the destination code in binary. When the ith high-order bit of the destination code is a 0, a 2 × 2 switch at level i connects the input to the upper output. Otherwise, the input is directed to the lower output.

In this network, it can be observed that, there is exactly one path from each source to any particular destination. There are some request patterns which cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 111.

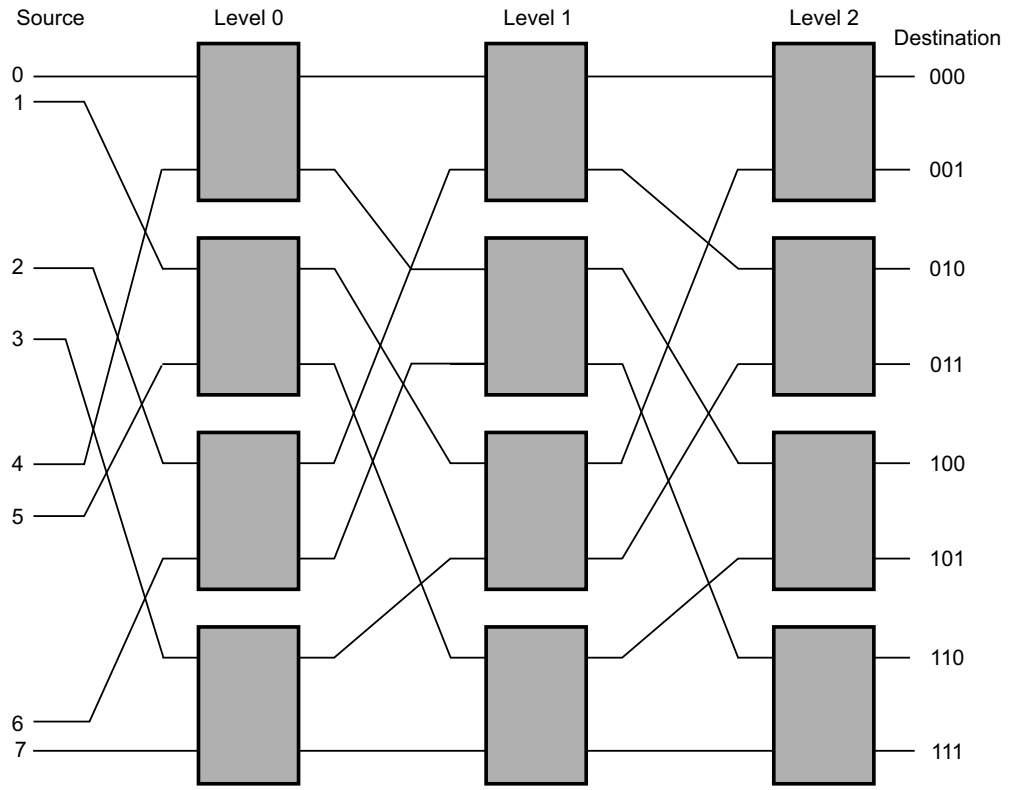
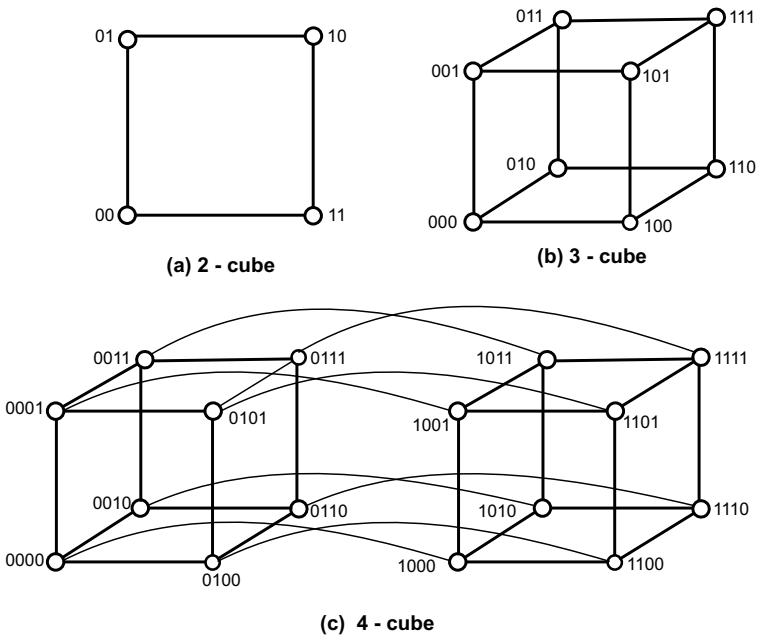


Fig. 5.10.15 8 × 8 Omega switching network

Hypercube Interconnection

The hyper cube interconnection is a binary n-cube architecture. With this interconnection it is possible to connect 2^n processors. Each processor form a node of the cube. It is not necessary that at each node there is a processor; however the node can have local memory module or I/O interface. The processor at one node has a direct communication path to n other nodes. Since there are 2^n nodes, there are 2^n distinct n-bit binary addresses.

The Fig. 5.10.16 shows the hyper cube structure for n = 2, 3 and 4. A two cube structure has 4 nodes, three cube structure has 8 nodes and 4 cube structure has 16 nodes. As shown in the Fig. 5.10.16, a 4 cube structure is formed by interconnecting two 3 cubes. The Fig. 5.10.16 also shows the binary addresses assigned to different nodes. They are assigned such that the addresses of two neighbours differ in exactly one bit position. For example, the three neighbours of the node with address 000 in the 3-cube structure are 001, 010 and 100. Each of these binary numbers differ from address 000 by one bit value.



(a) 2 - cube

(b) 3 - cube

(c) 4 - cube

Fig. 5.10.16 Hypercube structures

In n-cube structure, we require 1 to n links to route the message from one node to another. For example, in 3-cube structure we require one link to communicate from node 000 to 010, two links to communicate from node 000 to 101 and three links to communicate from node 000 to 111. We can calculate the number links require to communicate from one node to another. This is done by exclusive-ORing the binary

value of source node with the binary value to destination node. The number of 1s in the result are the number of links required for the communication. For example, in a 3-cube structure, to transfer message from node 001 to node 100 we require two links because exclusive-ORing of 001 and 100 is 101 and result has two ones.

5.10.5 Contention Problems in Multiprocessor Systems

A shared memory multiprocessor system consists of a number or processor sharing memory modules through an interconnection network. One of the main problems with this organization is how to avoid performance degradation due to memory sharing. The need for memory access by different processors give rise to contention which cause loss in performance. This contention is of three types :

1. Memory contention 2. Communication contention 3. Hot spot contention

5.10.5.1 Memory Contention

A memory module can handle only one access request at a time. Hence when several processors request the same memory module, it causes memory contention. This contention is resolved by an arbiter by permitting the requesters to proceed one at a time. However, the processors that wait to complete their memory cycle waste time in busy waiting. This results in loss of performance.

5.10.5.2 Communication Contention

When several requesting processors cannot complete their access due to the limitation of the interconnection network, the contention occurs and such contention is called **communication contention**.

This may happen even if the processors request access to different memory modules. Since the processor memory traffic is proportional to the number of processors, the bandwidth of the interconnection network must be proportional to the number of processors to avoid severe communication contention. Hence for systems containing a large number of processors, the interconnection network can become complex and expensive communication contention can result if several processors request for a common link in the interconnection network.

5.10.5.3 Hot Spot Contention

When several processors repeatedly access the same memory location, it gives rise to **hot spot contention**. This is because it creates hot spot in either the memory module or the interconnection network that get overloaded with the requests and create a bottle neck for system performance.

5.10.5.4 Techniques for Reducing Contentions

1. Local Memories
2. Better Interconnection Network
3. Cache Memory
4. Memory Allocation

1. Local Memories : To reduce contention, memory can be partitioned into global memory and local memory. Global memory can be accessed through interconnection network only. On the other hand, local memory is attached to processor so that it is directly accessed by the processor reducing network traffic.

Local memory can be divided further into shared and private memory. Shared memory is accessible by all processors, whereas a private memory is accessible only by the processor to which it is attached.

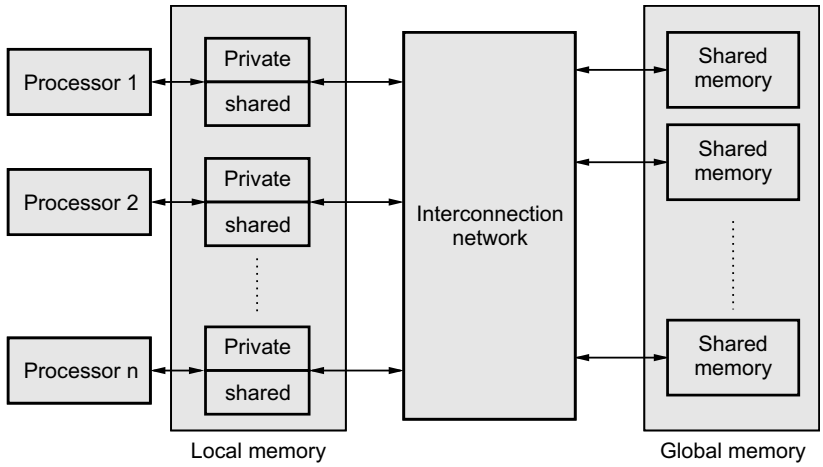


Fig. 5.10.17

Global memory modules are implicitly shared whereas a private memory is implicitly local.

Three types of memories

1. Local-private
2. Local-shared
3. Global shared

By allocating private data to private memory and shared data to the local-shared memory of the processor the traffic through the interconnection network and hence of contention can be reduced.

2. Better Interconnection Network : By using a high bandwidth network such as cross-bar switch, multiport memories or a multistage network, the communication contention on can be reduced. In a bus based system, techniques such as pipelining of several memory requests can be introduced to take full advantage of available bandwidth. The techniques of **combining** is used to reduce performance degradation due to hot-spot contention.

3. Cache Memory : Memory and communication contention can be reduced by attaching a private cache to each processor. Cache organization helps a lot in reducing memory traffic.

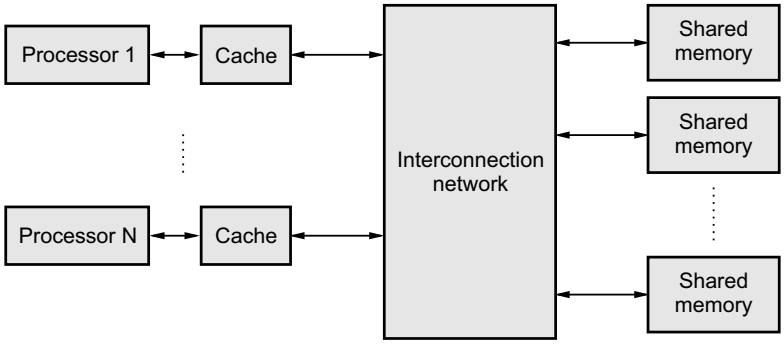


Fig. 5.10.18

4. Memory Allocation : Even if interconnection network meets the bandwidth requirements of the processor memory traffic, it is essential to consider how data structures are allocated to the shared memory modules. To minimize the contention, local memories should store code and data structures private to the processors to which they attach.

For shared data structures, the allocation should be uniformly distributed across the shared memory requests which result in uniform distribution of memory requests, thus reducing memory contention. For this the compiler should do proper allocation of data structures.

5.10.6 Mutual Exclusion

Because more than one processor shares the system memory and I/O devices through a common system bus, extra logic, must be added to ensure that only one processor has access to the system bus at a time. When one processor starts executing the program which uses the common resources, another processor is not allowed to access the used resources. This program region where the common resources are used is called critical program region. One can restrict another processor form accessing the used resources by using a techniques, called Mutual exclusion. In this technique, a binary flag called semaphore is stored in the shared memory to indicate where the shared memory is free

(semaphore = 1) to be accessed or busy (semaphore = 0). Testing and setting the semaphore is a critical operation and it must be performed by one processor at a time. This mutual exclusion technique can be implemented in 8086, 68000 and many other processors which support the multiprocessor environment.

5.10.7 Dead Lock

When two or more processors need more than one resources at a time. It may happen that one processor gets the access of first resource and another gets the access of second resource. But as both the resources are required to continue operation in both the processors, none of the processor can proceed further. Such situations are referred to as dead locks.

Dead locks can be avoided

1. By linking multiple resources together under one semaphore. So that linked resources can be accessed in a single action.
2. By setting the priorities for accessing the resources.

Review Questions

1. *Draw and explain the three-stage 8×8 omega multistage interconnection network.* **May-11, Marks 6**
2. *Draw and explain typical multiprocessor organisation.*
3. *Draw and explain non hierarchical loosely coupled multiprocessor system.*
4. *Draw and explain tightly coupled microprocessor system without private cache.*
5. *Draw and explain tightly coupled microprocessor system with private cache.*
6. *Explain various processor characteristics for multiprocessor.*
7. *Explain various physical forms available for the interconnection network.*
8. *Draw and explain the time shared bus.*
9. *Draw and explain the crossbar switch system organisation for multiprocessor.*
10. *Draw and explain the functional structure of crosspoint in a crossbar network.*
11. *Write a short note on multiport memory system.*
12. *Write a short note on contention problems in multiprocessor system*

5.11 Fault Tolerance **Dec.-03, 11**

In any software based system, there may be two types of failures. These are hardware failures and software failures (program errors). Fault tolerance is defined as the ability of a system to execute specified algorithms correctly regardless of these failures. Fault tolerance is an important design goal in many applications such as spacecraft control and telephone switching.

Hardware Failures : Majority hardware failures are because of physical causes such as component failure or electromagnetic interference. We can determine the type or nature and number of occurrences or frequency of these failures with the help of experiments. By performing analysis and using simulation models, it is possible to study the faults and their consequences.

Software Failures : Software faults are basically because of incorrect logic i.e. incorrect algorithm or incorrect programming i.e. design errors. So it is very much difficult to deal with such type of failures.

5.11.1 Redundancy

The concept of redundancy is very much related to fault tolerance. In a system, when any component fails, its function/s must be performed by any other component within the system which is fault-free. If such components are used just to take over duties of failed components which improves the reliability of the system and there is no significant effect on the computing performance, then they are called **redundant**. Let us see various types of redundancies.

1. **Hardware redundancy :** There are multiple copies of critical hardware units. When any hardware unit is failed, it is replaced by the similar another unit.
2. **Software redundancy :** There are multiple versions of programs for critical operations. When there is programming error, the program is replaced by another version.
3. **Information redundancy :** In a system, error-detecting or error-correcting codes are used.
4. **Time redundancy :** In this type, certain critical operations are repeated or retried.

5.11.2 Fault Tolerant Systems

There is presence of an error in a system due to mistake in design or due to failure of component. The above redudancies are useful to prevent these errors, and hence to prevent system failure. This section explains the use of redundant hardware to achieve fault tolerance. We can broadly classify fault tolerant systems as

- Static redundancy
- Dynamic redundancy.

5.11.2.1 Static Redundancy

In this technique, whenever there is hardware or software failure, the redundant hardware or software component is used. Thus the failed component is replaced by another new component which then becomes a permanent part of the system.

n-modular redundancy (nMR) : In this technique, n independent copies of the same unit are used in parallel. The outputs from all units are applied to a circuit called as 'voter'. The voter produces the output a word X. The voter is designed to output the value of X appearing on the majority of its n input buses. If an error is produced by any unit, voter takes care of masking that unit, provided more than half of the units produce the correct X values at all times. Thus the system formed with n identical units and a voter is said to employ n-modular redundancy (nMR).

When n = 3, the resulting version is called 'triple modular redundancy'. This is the frequently implemented version of nMR. Fig. 5.11.1 shows the example of triple modular redundancy (TMR).

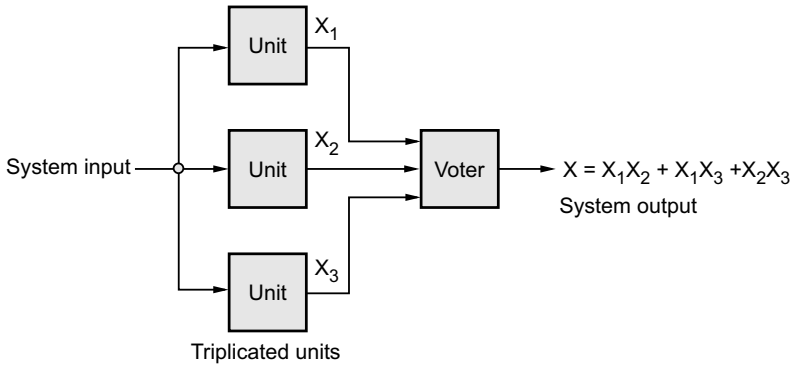


Fig. 5.11.1 Triple modular redundancy

The output of this system is given by logic equation,

$$X = X_1X_2 + X_1X_3 + X_2X_3$$

[+ → OR operation].

The output given by voter, X is always correct, assuming that no more than one of X_1, X_2, X_3 is incorrect and voter always produces error free output. If the output produced by any unit is faulty, then that is masked by using another unit. Thus a TMR system tolerates faulty behaviour by any one of its triplicated units.

Generally static redundancy is implemented at the processor level. At this level, the replicated units are CPUs, memory units, switching networks or entire computers. When CPUs are replicated, the resulting system is a type of multiprocessor.

5.11.2.2 Dynamic Redundancy

Unlike static redundancy, here the functions of the faulty unit are transferred to one or more fault-free units. Thus the faults are tolerated by reorganizing the system. This is usually done by performing following steps :

1. **Fault diagnosis :** In this the fault is detected by using special diagnostic procedures and then it is isolated to a replaceable or repairable unit.
2. **Fault elimination :** For removing the fault, one of the following methods is used,
   a. Repair the faulty unit,
   b. Replace the faulty unit by a spare unit,
   c. Logically reconfigure the system around the fault.
3. **Recovery :** In this step, the procedures are executed to restore the system to a state that existed before the occurrence of the fault, and from that point the normal operation is resumed.

Fig. 5.11.2 shows the example of dynamic redundancy.

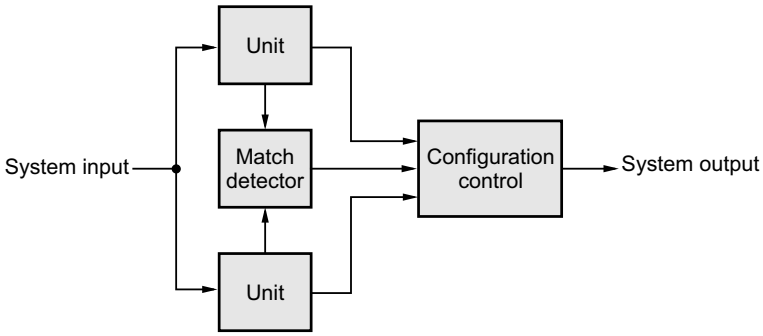


Fig. 5.11.2 Duplex system

As shown in Fig. 5.11.2, in this system two identical units are used which are non-redundant. Thus the system is duplex. These two units perform the same operations on the same data at the same time. The results generated by the two units are continuously compared by a circuit called a match detector or equality checker. When there is a mismatch between the outputs given by two units, this is detected by match detector. This is indicated as an occurrence of a fault. Thus when a fault is detected, the normal operation is temporarily suspended and the fault is identified by executing a testing procedure. Using this procedure, the faulty unit is identified and disconnected from the system, either physically or logically. After removing the faulty unit, the system can then be restarted in simplex mode. But now only fault-free unit is used in the system. The faulty unit is then repaired separately and eventually restored to the system.

Thus dynamic redundancy is more complex than static redundancy. But in dynamic redundancy faulty units are rapidly eliminated from the system. In static redundancy the functions of faulty units are masked by using redundant units and hence faults remain undetected until a total system failure occurs.

5.11.3 Redundant Disk Arrays

In computers, for providing large amounts of storage at low cost per bit disk memories (magnetic and optical) are used as secondary memory systems. However they have several drawbacks such as

1. Their data-transfer rates are relatively slow.
2. There is possibility of transient and catastrophic failures because of their electromechanical construction.

These draw-backs can be removed by using a disk memory built from an array of small disk units. Since all the disk units operate in parallel, data-transfer rate is increased. If there are n such parallel units, the effective data-transfer rate is n times that of single unit. Also, using an array of redundant disk units, fault tolerance is improved. By considering these issues a new system is designed referred to as RAID (Redundant Arrays of Inexpensive Disk). We have already discussed this in detail in chapter 4.

5.11.4 Fault Tolerance Measure

The ability of a system to tolerate faults can be measured in several ways. Two commonly used ways are availability and reliability.

5.11.4.1 Availability

It is defined as the fraction of its operating lifetime during which the system is not disabled by faults.

5.11.4.2 Reliability

For estimating reliability of a unit, a large number of samples of the unit are taken and their failure statistics is analysed. Reliability is defined as the probability of a unit or system surviving (functioning correctly) for a period of duration t. The failure rate is the fraction of the samples that fail per unit time. Typical variation of failure rate with time is shown in Fig. 5.11.3. The curve is like 'bathtub'.

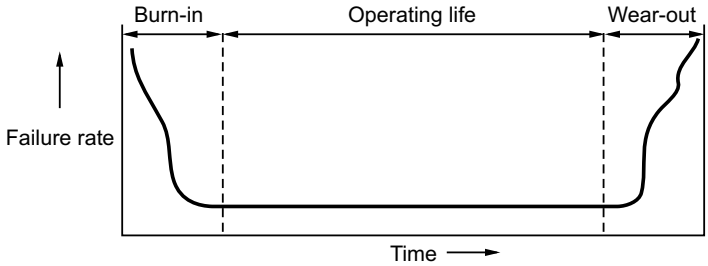


Fig. 5.11.3 Typical variation of failure rate with time

As shown in Fig. 5.11.3, failure rates can be observed with time by dividing time in parts as

1. **Burn-in period :** This is the early life of the unit. A high failure rate is experienced during this period. This reflects faults which occur during manufacturing or installation of the unit.
2. **Operating life :** During most of the unit's working life, failures occur randomly, but at a fairly constant rate. This period is the flat central part of the curve.
3. **Wear-out period :** Again a high-failure rate is observed during the period which is the end of the unit's life.

The reliability of a computer system can be studied using analytic approaches based on probability theory. Suppose the N(0) copies of a unit (e.g. CPU) begin their operating life at time t = 0. If N(t) are the number of units surviving after time t, then the number of failed units can be obtained as

$$N_f(t) = N(0) - N(t) \quad \ldots (5.11.1)$$

The reliability R(t) of the unit can be obtained as

$$R(t) = \frac{N(t)}{N(0)} \quad \ldots (5.11.2)$$

If λ is the unit's failure rate, it is assumed to be constant. Then the number of units dN_f that fail during the small interval of time from t to t + dt is given by,

$$dN_f = \lambda N(t)\, dt \quad \ldots (5.11.3)$$

From equation (5.11.1), we have

$$N(t) = N(0) - N_f(t)$$

and N(0) is independent of t.

Hence,

$$dN = - dN_f \quad \ldots (5.11.4)$$

Substituting value of dN_f in equation (5.11.3), we have

$$dN = - \lambda N(t)\, dt \quad \ldots (5.11.5)$$

Using equations (5.11.2) and (5.11.5) we can write,

$$dR = \frac{dN}{dN(0)}$$

$$\therefore \quad dR = \frac{-\lambda\, N(t)\, dt}{N(0)}$$

$$\therefore \quad \frac{dR}{dt} = -\lambda R(t)$$

By taking integration with the boundary value R(0) = 1 gives,

$$R(t) = e^{-\lambda t} \quad \ldots (5.11.6)$$

This is called **exponential law** of failure. This is oftenly used to model the reliability of the components in a computer system.

5.11.4.3 System Reliability

We can calculate the reliability of the entire system by knowing the failure rates of individual units of the system.

Fig. 5.11.4 gives two basic reliability structures, series and parallel.

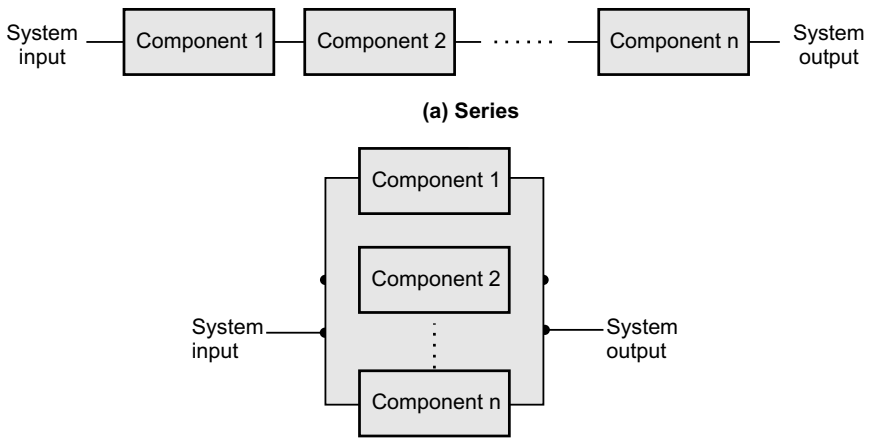


Fig. 5.11.4 Basic reliability structures

1. Series structure : In a series system, output of first component acts as input for second, output of second as input for third and so on. So if any component fails, the entire system fails. Hence, the system reliability can be obtained by multiplying the reliabilities of the individual components.

If there are n components connected in series, then system reliability R can be expressed as

$$R = R_1 \times R_2 \times R_3 \times \ldots \times R_n = \prod_{i=1}^{n} R_i$$

where R_1, R_2, R_3, …, R_n are the reliabilities of component 1, component 2, component 3, … , component n respectively.

2. Parallel structure : In a parallel system, if all the components fail, then only the system fails. Hence the system's unreliability F = 1 – R is the product of the component unreliabilities $1 - R_i$. So we can express the reliability of the system as,

$$R = 1 - \prod_{i=1}^{n} (1 - R_i)$$

From all above discussion, we can conclude that series structure decreases reliability while parallel structure increases reliability. Thus a parallel connection of n units is a basic fault tolerant structure.

This is not always the case that the system structure is completely series or completely parallel. Sometimes there is a combination of series and parallel structures. In this case, systems can be decomposed into series and parallel subsystems. Their reliability can be calculated by repeated application of reliability equations for both types of structures.

For example, consider the system shown in the Fig. 5.11.5.

The system is series-parallel system. This consists of two subsystems, S_1 and S_2. The subsystems S_1 and S_2 are connected in series. The S_1 and S_2 themselves are parallel systems.

If the individual component has reliability R, then the overall system reliability can be obtained as,

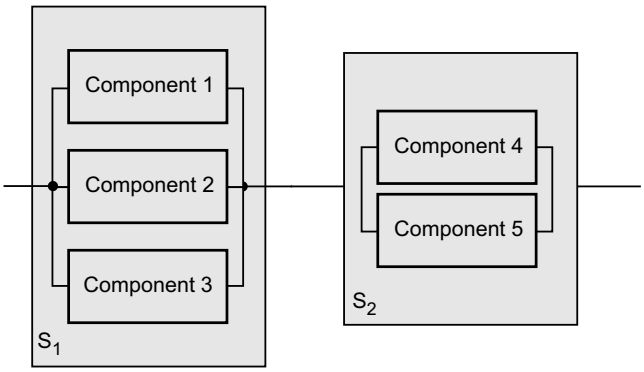


Fig. 5.11.5 Series parallel system

$$R(S) = [1 - (1 - R)^3] [1 - (1 - R)^2] = 6R^2 - 9R^3 + 5R^4 - R^5$$

5.11.4.4 Mean Time To Failure (MTTF)

This is a useful measure of the expected working life of a unit. MTTF can be expressed as,

$$MTTF = \int_0^{\infty} t\, f(t) dt$$

Here, $f(t) = \frac{dF(t)}{dt}$ where F(t) is the unreliability of a system i.e. 1 – R(t).

From equation (5.11.6) we can write,

$$MTTF = \int_0^{\infty} t \lambda e^{-\lambda t} dt = \frac{1}{\lambda}$$

Thus we can say that the expected working life of a unit with an exponentially distributed reliability is the reciprocal of its failure rate.

5.11.4.5 Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR)

Uptil now we have seen fault tolerance measurement by methods which do not consider maintenance or repair activities. If the repair process makes the system 'as good as new', then the average duration between the completion of a repair and the occurrence of the next fault is the system's Mean Time To Failure (MTTF). Mean Time To Repair (MTTR) is the expected time between system failure and the completion of repair. The total of MTTF and MTTR is referred to as the mean time between failures (MTBF). When MTTR is very small, MTBF is approximately equal to MTTF. The expected availability, A of the system can be obtained using formula,

$$A = \frac{MTTF}{MTTF + MTTR} \quad \ldots (5.11.7)$$

$$\therefore \quad A = \frac{MTTF}{MTBF}$$

From equation (5.11.7) it is cleared that the availability of the system can be increased either by increasing the system's reliability (indicated by MTTF) or by reducing the time needed to repair after the occurrence of a fault.

Thus the fault tolerance measurement by reliability analysis considered only static systems in which there is no maintenance or repair activities whereas MTTF, MTTR methods are applicable for dynamic systems.

5.11.5 Fault Tolerant Computers

Let us see the examples of fault tolerant computers.

5.11.5.1 The Tandem Nonstop

It is a commercial fault-tolerant multiprocessor series. The NonStop I is the first model from this series. Its specifications are as given below :

1. This is a small-scale multiprocessor based on bipolar (TTL) MSI integrated circuit technology.
2. Its CPU is a custom-designed 16-bit processor of the CISC type, hardwired, stack-oriented organization.
3. Operating frequency is 10MHz.
4. CPU performance is 0.7 MIP.
5. Virtual memory is of 512kB.
6. System contains 2 to 16 processors.

5.11.5.2 The Tandem VLX

This is the advanced model whose improved specifications are described below :

1. This is based on bipolar (ECL) gate array integrated circuit technology.
2. A 32-bit microprogrammed architecture.
3. The speedup techniques such as pipelining and 64 kB unified cache are used.
4. Operating frequency is 12 MHz.
5. CPU performance is 3.0 MIPS.
6. Virtual memory is of 1 GB.

5.11.5.3 The Tandem Himalaya

This is the next advanced model. Its important specifications are listed below.

1. It uses CMOS technology.
2. The CPU used is 64-bit of the RISC type.
3. This superscalar microprocessor supports a two-level cache and a virtual memory of 2^{64} B.
4. Operating frequency is 200 MHz.
5. CPU performance is 100 MIPS.
6. These systems are designed in two or four processor clusters built around a high-speed shared bus.
7. The massively parallel systems contain hundreds of processors. A mesh-like structure is constructed by linking clusters together via a large-scale interconnection network.

5.11.5.4 Architectural Details of Tandem's NonStop Computers

Applications of on-line transaction processing such as on-line ticket reservation have very high performance requirements.

The objectives of Tandem's "NonStop" architecture are as follows.

1. Any one hardware fault is prevented by a system organization. That is a single-point failure is prevented.
2. There is dynamic on-line detection of faults. Faulty units are removed from the system for repair, and redundant components keep the system in operation. After repairing of faulty units, they are returned to the system.
3. Processor, memory and IO capacity is increased. The scalability allows this without affecting the application's software.

To achieve all these objectives, multiprocessor architecture is used in these systems. The redundancy provided by multiple processors is useful both for fault tolerance and for high performance. The single point failures are masked by using duplicated units of power supply, system bus and IO controllers. For error detection and correction, standard coding techniques are used. The NonStop operating system Kernel is built around duplex, distributed processes. This exchanges messages for interprocess communication. Hard disks and other IO devices are connected to two IO controllers.

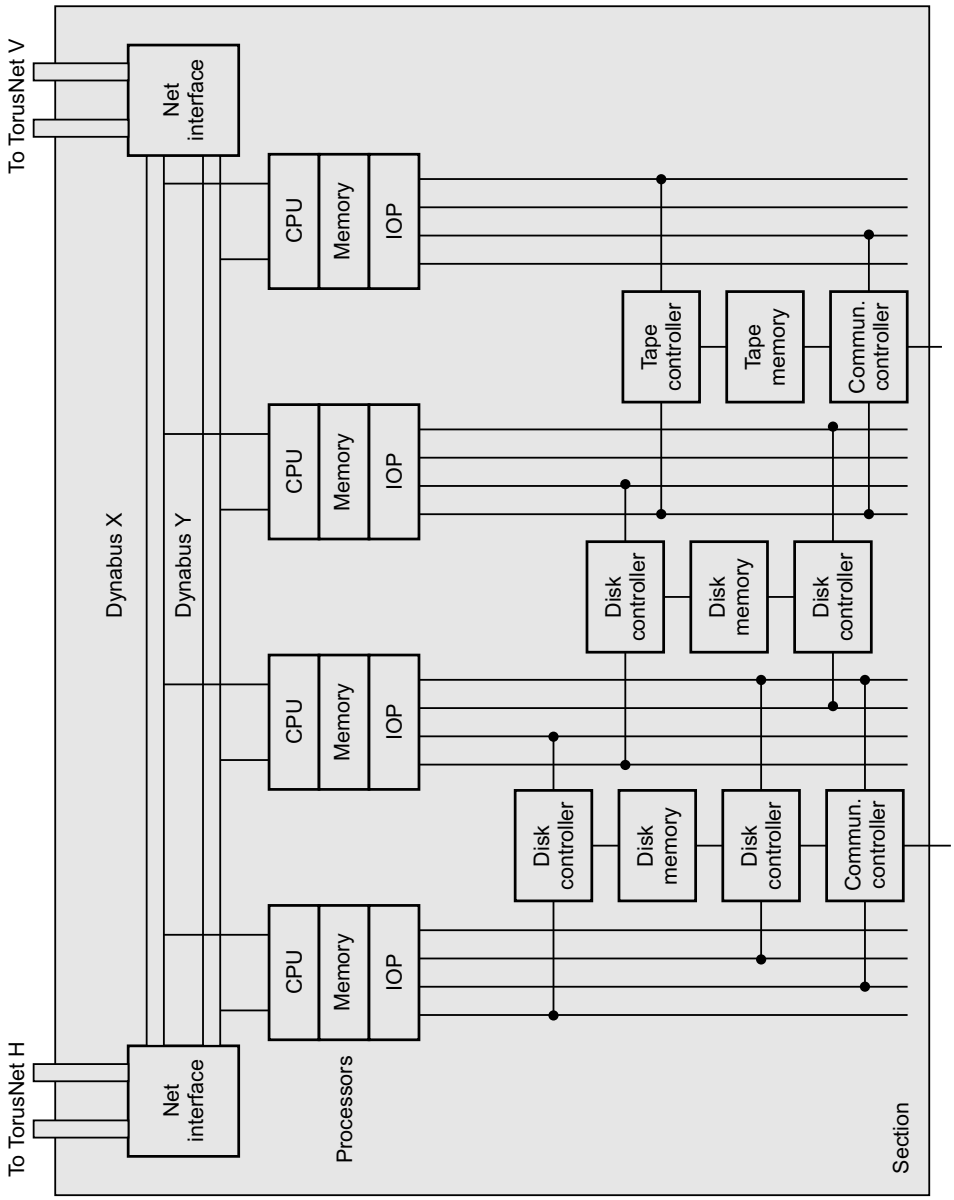


Fig. 5.11.6 Processor cluster (section) in Tandem Nonstop computers

One of the IO controllers owns each device. Operating system switches this ownership at any time. Each IO device is controlled by the software. This software is present in a redundant primary or backup pair of processes. Two functions are performed by this primary process. First is, it manages the device and second is, it sends 'checkpoint' information to the backup process to keep it up-to-date. Using similar way, users processes are handled. When one process is executing on one processor, a backup copy of the same process is automatically started executing on another processor.

The basic hardware structure of every NonStop system is shown in Fig. 5.11.6. This is the structure of four-processor cluster or section. As shown in Fig. 5.11.6, there are four processors. Each processor contains a CPU, a portion of main memory and an IO processor. There is a high-speed interprocessor bus called as 'dynabus'. This is duplicated. The processors in a section communicate with one another via dynabus. There is a set of IO buses called as channels. The channel links each processor to a set of IO controllers because of which every IO controller gets connected to two processors. A loosely coupled system containing tens or hundreds of clusters can be formed. The sections in the system can communication via a LAN-style network. These type of configurations are well suited to OLTP servers. A large number of independent tasks can be performed using such configurations.

5.11.5.5 Architectural Details of the Tandem Himalaya Computers

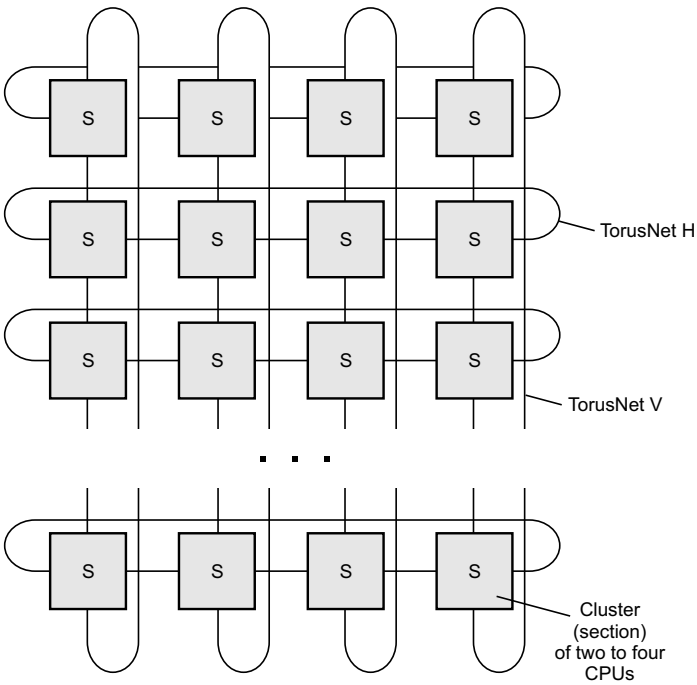


Fig. 5.11.7 Toroidal interconnection network of the Tandem Himalaya computer

We have already seen the specifications if Tandem Himalaya computers. Each Himalayas CPU actually contains two R4400s operating in lockstep, with one processor (the slave) serving as a check on the other (the master). A novel type of interconnection network is also introduced by Himalaya. This network is called **TorusNet.** This network uses fiber-optic cables to connect the clusters. These clusters or sections can be linked together in a ring network. Each cluster is participated in separate H (horizontal) and V(vertical) rings. The TorusNet H and V rings accommodate up to 4 and 14 clusters respectively. The 56 clusters (i.e. $54 \times 4 = 224$ processors) can be connected in this way. The interconnected network has toroidal structure as shown in Fig. 5.11.7. In this structure, every cluster is directly linked to four others and indirectly to all the clusters in the system.

In this large Himalaya system, many alternative paths among its processors are provided. So the system can tolerate the simultaneous failure of several of its clusters and their interconnections.

Review Questions

1. *Explain reliability and MTTF.* **Dec.-03, Marks 2**
2. *Give a brief account on fault tolerance.* **Dec.-03, Marks 8**
3. *Define fault tolerance. How is it related to redundancy ? Explain different approaches for designing fault tolerance systems.* **Dec.-11, Marks 8**
4. *What do you mean by static redundancy ?*
5. *What do you mean by dynamic redundancy ?*
6. *What is n-modular redundancy ?*
7. *Give the difference between static and dynamic redundancies.*
8. *State any four ways to measure fault tolerance.*
9. *What is system reliability ? Draw the basic reliability structures for series and parallel systems and give expressions to obtain reliabilities for them.*
10. *What is MTTR ?*
11. *Give any one example of fault tolerant computer. Give specifications and architectural details of it with neat diagram.*

Two Marks Questions with Answers

Q.1 ***Define interface.***

Ans. : The word interface refers to the boundary between two circuits or devices.

Q.2 ***What is the necessity of an interface ? OR What are the functions of a typical I/O interface ?*** **Dec.-06, 07**

Ans. : An interface is necessary to coordinate the transfer of data between the CPU and external devices. The functions performed by an I/O interface are :

1. Handle data transfer between much slower peripherals and CPU or memory.
2. Handle data transfer between CPU or memory and peripherals having different data formats and word lengths.
3. Match signal levels of different I/O protocols with computer signal levels.
4. Provides necessary driving capabilities - sinking and sourcing currents.

Q.3 ***What is an I/O channel ?***

Ans. : An I/O channel is actually a special purpose processor, also called peripheral processor. The main processor initiates a transfer by passing the required information in the input output channel. The channel then takes over and controls the actual transfer of data.

Q.4 ***Name the two interfacing techniques***

Ans. : I/O devices can be interfaced to a computer system I/O in two ways :

- Memory mapped I/O
- I/O mapped I/O

Q.5 ***What is memory mapped I/O ?***

Ans. : The technique in which the total memory address space is partitioned and part of this space is devoted to I/O addressing is called memory mapped I/O technique.

Q.6 ***What is I/O mapped I/O ?***

Ans. : The technique in which separate I/O address space, apart from total memory space is used to access I/O is called I/O mapped I/O technique.

Q.7 ***Why program controlled I/O is unsuitable for high-speed data transfer ?***

Ans. : In program controlled I/O, several program instructions have to be executed for each data word transferred between the external devices and memory and hence program controlled I/O is unsuitable for high-speed data transfer.

Q.8 ***What is programmed I/O ?***

Ans. : I/O operations will mean a data transfer between an I/O device and memory or between an I/O device and the processor. If in any computer system I/O operations are completely controlled by the processor, then that system is said to be using programmed I/O.

Q.9 ***What is an interrupt ?*** **Dec.-09**

Ans. : An interrupt is an event that causes the execution of one program to be suspended and another program to be executed.

Q.10 *How does the processor handle an interrupt request ?* **May-07, Dec.-07**

Ans. : Processor identifies source of interrupt. Processor obtains memory address of interrupt handler. PC and other processor status information are saved. PC is loaded with address of interrupt handler and program control is transferred to interrupt handler.

Q.11 *Why are interrupt masks provided in any processor ?* **May-06**

Ans. : In the processor those interrupts which can be masked under software control are called **maskable interrupts**. Once the interrupt is masked, the processor is not interrupted even though interrupt is activated. This facility is necessary when processor is executing critical program which should not be interrupted or it may be executing time related function.

Q.12 *What is an non-maskable interrupt ? What is the action performed on receipt of a NMI ?* **Dec.-08**

Ans. : The interrupts which can not be masked under software control are called non-maskable interrupts. Following action is performed on receipt of a NMI.

- Processor obtains memory address of interrupt handler of NMI
- PC and other processor status information are saved.
- PC is loaded with address of interrupt handler of NMI and program control is transferred to interrupt handler.

Q.13 *What are vectored interrupts ?* **May-09**

Ans. : If the processor has predefined starting address for interrupt service routine of an interrupt then that address is called **vector address** and such interrupts are called **vector interrupts**.

Q.14 *What do you mean by interrupt nesting ?*

Ans. : An interrupt which interrupts the currently executing interrupt service routine for another interrupt is called nested interrupt. A system of interrupts that allows an interrupt service routine to be interrupted is known as interrupts nesting system.

Q.15 *What is the advantage of using interrupt initiated data transfer over transfer under program control without interrupt ?* **May-07**

Ans. : Refer section 5.4.

Q.16 *What is DMA ? or What is DMA operation ? State its advantages or Why do we need DMA ?* **Dec.-06, 07**

Ans. : A special control unit may be provided to enable transfer a block of data directly between an external device and memory without contiguous intervention by the CPU. This approach is called DMA (Direct Memory Access). The data transfer using such approach is called DMA operation.

The two main advantages of DMA operation are:

- The data transfer is very fast.
- Processor is not involved in the data transfer operation and hence it is free to execute other tasks.

Q.17 *Specify the different types of the DMA transfer techniques ?*

Ans. : The different types of the DMA transfer techniques are :

- Single transfer mode(cycle-stealing mode)
- Block Transfer Mode(Brust Mode)
- Demand Transfer Mode

Q.18 *What are the three types of channels are usually found in large computers ?* **Dec.-07**

Ans. :

- DMA channel
- Selector I/O channel
- Multiplexer I/O channel

Q.19 *What are the necessary operations needed to start an I/O operation using DMA ?* **Dec.-07**

Ans. : When the CPU wishes to read or write a block of data, it issues a command to the DMA module or DMA channel by sending the following information to the DMA channel/controller :

1. A read or write operation.
2. The address of I/O device involved.
3. The starting address in memory to read from or write to.
4. The number of words to be read or written.

Q.20 *What is a bus ?*

Ans. : A collection of wires that connects several devices is called a bus.

Q.21 *Define word length.*

Ans. : Each group of n bits is referred to as a word of information and n is called the word length.

Q.22 *What are the different types of buses ?*

Ans. :

- Synchronous bus
- Asynchronous bus

Q.23 *Define synchronous bus.*

Ans. : The bus in which all devices connected to the bus derive timing information from a common clock signal called synchronous bus.

Q.24 *Define Asynchronous bus.*

Ans. : The bus in which the common clock is eliminated and data transfer on the system bus is achieved by the use of a hand shake between the processor and the device being addressed is called asynchronous bus.

Q.25 *What is meant by bus arbitration ?* **May-10**

Ans. : Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of bus master is usually done on the priority basis.

Q.26 *State the approaches used for bus arbitration.*

Ans. : There are two approaches used for bus arbitration : Centralized and distributed.

Q.27 *State different arbitration schemes that use the centralized bus arbitration approach.*

Ans. : There are three different arbitration schemes that use the centralized bus arbitration approach. These schemes are :

a. Daisy chaining

b. Polling method

c. Independent request

Q.28 *What are tri-state gates ?* **May-08**

Ans. : The gates having three output states : logic 0, logic 1 and high-impedance are called tri-state gates.

Q.29 *Name and give the purpose of widely used bus standard.* **May-10**

Ans. : Universal serial bus (USB)

Q.30 *Name any three of the standard I/O interface.*

Ans. :
- SCSI (small computer system interface),bus standards
- Back plane bus standards
- IEEE 796 bus (multi-bus signals)
- NUBUS
- IEEE 488 bus standard

Q.31 *What is NUBUS ?*

Ans. : A NUBUS is a processor independent, synchronous bus standard intended for use in 32-bit microprocessor system. It defines a backplane into which up to 16 devices may be plugged each in the form of circuit board of standard dimensions.

Q.32 *What are the two important mechanisms for implementing I/O operations?*

Ans. : There are two commonly used mechanisms for implementing I/O operations. They are interrupts and direct memory access.

Q.33 *What is an exception?* **May-11**

Ans. : The term exception is often used to refer to any event that causes an interruption.

Q.34 *How does the processor handle an interrupt request ?* **May-07, Dec.-07**

Ans. : Processor identifies source of interrupt

Processor obtains memory address of interrupt handler

PC and other Processor status information are saved

PC is loaded with address of interrupt handler and program control is transferred to interrupt handler.

Q.35 *What do you mean by an interrupt acknowledge signal?*

Ans. : The processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This may be accomplished by an interrupt acknowledge signal.

Q.36 *What is interrupt latency?*

Ans. : Interrupt latency is the delay between an interrupt request is received and the start of execution of the interrupt-service routine.

Q.37 *What is known as cycle-stealing?*

Ans. : The processor originates most memory access cycles, the DMA controller can be said to "steal" memory cycles from the processor. Hence, this interweaving technique is usually called cycle stealing.

Q.38 *What is known as block/burst mode?*

Ans. : The DMA controller may be given exclusive access to the main memory to transfer a block of data without interrupt. This is known as block/burst mode.

Q.39 *What is called a bus master?*

Ans. : The device that is allowed to initiate data transfers on the bus at any given time is called the bus master.

Q.40 *What is known as distributed arbitration?*

Ans. : Distributed arbitration means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process, without using a central arbiter

Q.41 *What is a strobe?*

Ans. : Strobe captures the values of the data given instant and store them into a buffer.

Q.42 *What is meant by handshake?*

Ans. : Handshake is used between the master and the slave for controlling data transfers on the bus.

Q.43 ***What is a bit-rate?***

Ans. : The speed of transmission is known as a bit rate.

Q.44 ***What is a initiator?***

Ans. : A master is called an initiator in PCI technology.

Q.45 ***What is called a target?***

Ans. : The addressed device that responds to read and write commands is called a target.

Q.46 ***What is a transaction?***

Ans. : A complete transfer operation on the bus involving an address and a burst of data is called a transaction.

Q.47 ***What are known as asynchronous events?***

Ans. : The event of pressing a key is not synchronized to any other event in a computer system. The data generated by the keyboard are called asynchronous.

Q.48 ***What is called a hub?***

Ans. : Each node of the tree has a device called a hub which acts as an intermediate control point between the host and the I/O devices.

Q.49 ***What is a root hub?***

Ans. : At the root of a tree, a root hub connects the entire tree to the host computer.

Q.50 ***What is plug-and-play technology ?*** **May-11**

Ans. : The plug-and–play technology means that a new device, such as an additional speaker or mouse or printer, etc. can be connected at any time while the system is operating.

Q.51 ***Write down the CPU steps to determine the status of I/O device.*** **Dec.-11**

Ans. : The CPU steps to determine the status of I/O device are :

1. Read the I/O device's status bit
2. Test the status bit to determine if the device is ready to begin transferring data.
3. If not ready, return to step 1, otherwise, perform the data transfer.

Q.52 ***What is a tri-state circuit and what are the advantages of tri-state circuits ?***

Ans. : A tri-state circuit is a circuit whose output can be in the one of the three possible state : High, low and High-impedance. The high-impedance state denotes the

state of a line that is electrically disconnected from all voltage sources, that is, an open-circuited or floating line. The advantages of tri-state circuit are :

The greatly increase the fan-in and fan-out limits of bus lines, permitting very large number of devices to be connected to the same line.

They support bi-directional transmission over the bus by allowing the same bus connection to serve as an input port and as an output port at different times.

Q.53 *What is meant by bus arbitration ?* **May-10**

Ans. : If several devices can generate requests for bus access simultaneously, the bus master needs a way to select one of the devices; this selection process is called bus arbitration.

Q.54 *State the two control methods used in asynchronous communication.*

Ans. : The two control methods used in asynchronous communication are :

1. One-way control in which one of the two communicating devices supplies all timing signals.
2. Two-way or interlocked, control in which both devices generate timing signals.

Q.55 *What are the limitations of programmed I/O. ?* **Dec.-11**

Ans. : The limitations of programmed I/O are :

1. Programmed I/O requires that all I/O operations be executed under the direct control of the CPU.
2. In programmed I/O, I/O device does not have direct access to main memory.
3. Data transfer speed is low.

Q.56 *Write down the CPU steps to determine the status of I/O device.* **Dec.-11**

Ans. : The CPU steps to determine the status of I/O device are :

1. Read the I/O device's status bit.
2. Test the status bit to determine if the device is ready to begin transferring data.
3. If not ready, return to step 1; otherwise, perform the data transfer.

Q.57 *Draw the simple structure of an interface.*

Ans. : Refer section 5.3.1.1.

Q.58 *How do you handle simultaneous and multiple interrupts ?*

Ans. : Refer section 5.3.3.3.

❑❑❑

6 RISC, CISC, Superscalar and Vector Processors

Syllabus

RISC and CISC processors, Superscalar and vector processor.

Contents

6.1 RISC Processor

6.2 RISC Versus CISC ***May-10, 11,***

. . . . . ***Dec.-10,*** · · · · · · · · · · · · · · · *Marks 16*

6.3 RISC Properties

6.4 RISC Addressing Modes

6.5 RISC Evaluations

6.6 On-chip Register File Versus Cache Evaluation

6.7 Overview of RISC Development and Current Systems

6.8 Superscalar Processor . . . . . ***May-11,*** · · · · · · · · · · · · · · · *Marks 10*

6.9 Vector Processor . . . . . ***Dec.-10,*** · · · · · · · · · · · · · · · *Marks 16*

In this chapter we study RISC and superscalar processors and their organization.

6.1 RISC Processor

RISC is an acronym for Reduced Instruction Set Computer. It is intended as a contrast with CISC machines which are Complex Instruction Set Computers. Today RISC processors such as the Intel i860, SPARC, MIPS R3000, IBM RS/6000 etc. have faster clock rates. The clock rates range from 20 to 120 MHz determined by the implementation technology employed.

Most RISC processors use hardwired control. The architectural details of RISC processor are covered in next point which compares RISC with CISC. The most of the RISC processors use 32-bit instructions. They are very few instructions. The instructions are predominantly register-based. The limited (3 to 5) addressing modes are used by these processors. The memory access cycle is broken into pipelined access operations. This involves the use of caches and working registers. A large register file and separate instruction and data caches are used. This benefits internal data forwarding and eliminates unnecessary storage of intermediate results. Using hardwired control, the clock cycles per instruction (CPI) are reduced to 1 for most RISC instructions. This is the advantage of having all instructions of equal size.

Review Question

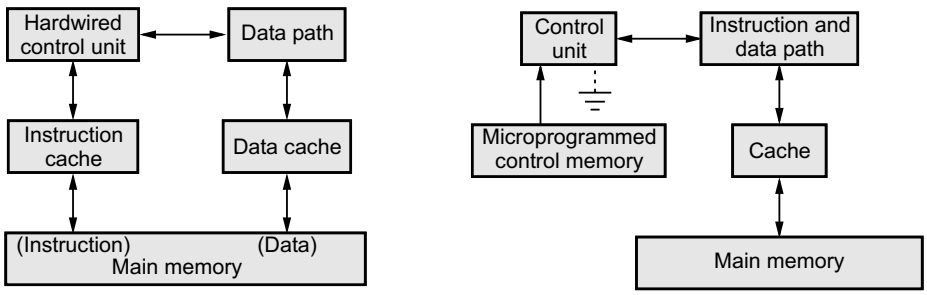
1. *What is RISC processor ?*

6.2 RISC Versus CISC

May-10, 11, Dec.-10

Future processors may be designed with features from both types. Therefore the boundary between RISC and CISC architectures has become blurred in recent years.

Fig. 6.2.1 shows the architectural distinctions between RISC and CISC.



(a) The RISC architecture with hardwired control and split instruction cache and data cache

(b) The CISC architecture with microprogrammed control and unified cache

Fig. 6.2.1

As shown in Fig. 6.2.1, RISC architecture uses separate instruction and data caches. Their access paths are also different. (Note : exceptions do exist). In a CISC processor, there is a unified cache for holding both, instructions and data. Therefore they have to share the same path for data and instruction.

The hardwired control is found in most RISC processors while the traditional CISC processors use microprogrammed control. Thus the control memory is needed in these CISC processors. This may significantly slow down the instruction execution. The modern CISC processors may also use hardwired control. So split caches and hardwired control are not exclusive in RISC machines.

Let us compare the characteristics of RISC and CISC processors. Table 6.2.1 shows the comparison between them.

No.	Characteristics	CISC	RISC
1.	Instruction size	Varies	Fixed
2.	Instruction length	1, 2, 3 or 4 bytes	4 bytes
3.	Number of Instruction	More	Less
4.	Instruction decoding	Serial (slow) to decode	Easy (quick) to decode
5.	Instruction sementics	Varies from simple to complex ; possibly many dependent operations per instruction	Almost always one simple operation
6.	Addressing Modes	Supports complex addressing modes.	Complex addressing modes are synthesized in software.
7.	Instruction execution speed	Slow (depend on complexity of instruction)	Medium
8.	Instruction execution	By microprogram. Complex instructions taking multiple cycles.	By hardware. Simple instructions taking one cycle.
9.	Registers	Few, may be special purpose	Many, general purpose
10.	Memory references	Combined with operations in many different types of instructions	Not combined with operations, i.e., load/store architecture
11.	Hardware	Complicated	Simple
12.	Hardware design focus	Take the advantage of microcoded implementations	Take the advantage of implementations with one pipeline and no microcode
13.	Memory access	Frequently	Rarely
14.	Instruction format	Field placement varies	Regular, consistent placement of fields

15.	Pipelined	Not pipelined or less pipelined.	Highly pipelined.
16.	Conditional jump	Conditional jump is usually based on status register bit.	Can be based on a bit anywhere in memory.
17.	Compiler	Simple	Complicated
18.	Examples	ARM, 8051, ATMEL, AVR, etc.	Intel X86, Motorola 68000 series.

Table 6.2.1 Comparison between RISC and CISC processors characteristics

Review Questions

1. *Compare RISC and CISC architectures.* **May-10, Marks 2**
2. *Explain the difference between CISC and RISC processors.* **Dec.-10, Marks 16**
3. *Distinguish between RISC and CISC processors.* **May-11, Marks 6**

6.3 RISC Properties

The designer of RISC architecture says that a complete overhaul is needed in the way we think about computer architecture and that nearly all conventional computers are architecturally complicated. They designed RISC architecture considering the properties which are discussed below.

Register to Register Operations

A large number of general purpose registers or the use of compiler technology to optimize register usage. It is the most important characteristics of RISC processor. This architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage. To implement register to register operations, RISC processors provide multiple sets of registers. These register sets are organized into overlapped windows and act as a small, fast buffer for holding a subset of all variables that are most likely to be used. Fig. 6.3.1 shows overlapping register windows. The window is divided into three fixed- size areas. **Parameter registers** hold parameters passed down from the procedure that called the current procedure and results to be passed back up. **Local registers** are used for local variables, as assigned by the compiler. **Temporary registers** are used to exchange parameters and results with the procedure called by current procedure. The temporary registers of current procedure are physically same as parameter registers of the *called* procedure. This overlap permits parameters to be passed without actual movement of data.

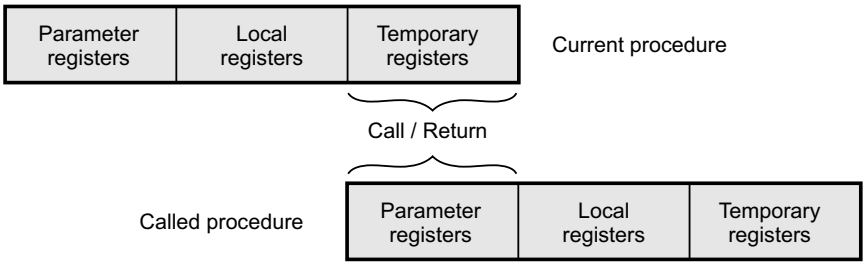


Fig. 6.3.1 Overlapping register window

One Instruction Per Cycle

In RISC processors, there is an one instruction per machine cycle. A machine cycle is defined to be the time it takes to fetch two operands from registers, performs an ALU operation, and stores the result in a register. Due to this, RISC machine instructions are not complicated and can execute about as fast as, microinstructions on CISC machines.

Hardwired Instructions

With simple, one cycle instruction, there is no need for microinstructions. The machine instructions can be hardwired. These instructions are executed faster than the instructions implemented with microinstructions, since it is not necessary to access a microprogram control memory during instruction execution.

Reduced Number of Instructions

RISC processor provides limited number of instructions, which simplifies the design of control unit.

Simple Addressing Modes

RISC processor uses simple addressing modes. Almost all instructions use simple register addressing. Several other addressing modes are also used, such as displacement and PC-relative. It does not support complexed addressing modes. They can be synthesized in software from the simple ones. This architecture of RISC processor simplifies the instruction set and the control unit.

Simple Instruction Format

RISC processor uses simple instruction formats with fixed instruction length. The instruction length is aligned on word boundaries. Field locations, especially opcodes are fixed.

This architecture provides following benefits :

- With fixed fields, opcode decoding and register operand addressing can occur simultaneously.
- It simplifies the design of control unit.
- Instruction fetching is optimized since word-length units are fetched.

Instruction Pipelining

Recall the concept of pipelining. The CPU contains several independent units that work in parallel. One of them fetches the instructions and other ones decode and execute them. At any instant, several instructions are in various stages of processing. All RISC processors provide this pipelining feature.

In RISC processor most instructions are register to register. These instructions have the following two phases :

- Instruction fetch (I)
- Execute (E)

The instruction fetch phase fetches the instruction to be executed from memory and then execute phase performs an ALU operation with register input and output to execute the instruction.

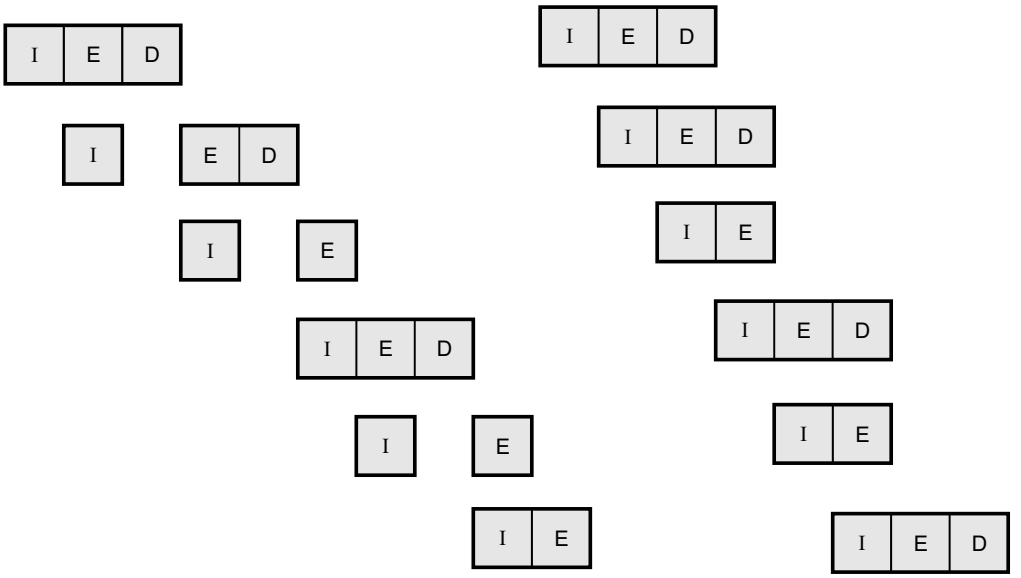
In case of load and store instructions, three phases are required :

- Instruction fetch (I)
- Execute (E)
- Data transfer (D)

Here, also the instruction fetch phase fetches the instruction to be executed. In execution phase address of memory is calculated and in data transfer phase actual data is transferred from register-to-memory or from memory-to-register depending upon the instruction.

The Fig. 6.3.2 shows that how these phases can be overlapped in RISC processors to achieve the pipelining. The Fig. 6.3.2 (a) shows the two-way pipelining and the Fig. 6.3.2 (b) shows the three way pipelining.

In two way instruction pipelining, I and E phases of two different instructions are performed simultaneously. In three way instruction pipelining, three instructions can be overlapped. Therefore, maximum execution rate in two way instruction pipelining is twice the normal execution rate and it is three times the normal execution rate in case of three way instruction pipelining.



(a) Two way instruction pipelining (b) Three way instruction pipelining

Fig. 6.3.2

Review Questions

1. *Explain the properties of RISC architecture.*
2. *Explain the advantages and disadvantages of RISC architecture.*

6.4 RISC Addressing Modes

The small instruction set of typical RISC processor consists mostly of register-to-register operations, and simple load and store operations for memory access. Each operand is brought into a processor register with a load instruction and results are transferred to memory by means of store instruction. In this architecture almost all instructions have simple register addressing, and thus it uses only a few addressing modes. Usually, the RISC processor has three basic addressing modes :

- Register addressing
- Immediate operand and
- Relative to PC addressing for branch instructions

Register addressing : In register addressing, the instruction usually consists of three fields : opcode field which specifies an operation, one or two source register fields and one destination register field. The operation is performed with the data specified in the source register fields and the result is stored in the destination register field.

For example : ADD R_1, R_2, R_3 ; $R_3 \leftarrow R_1 + R_2$. In case of memory access one source register specifies the memory address and second source register specifies the offset. For example, LD (R_1) R_2 ; $R_3 \leftarrow M [R_1 + R_2]$

Immediate operand addressing : In immediate operand addressing mode, the second source is an immediate operand. The operation is performed with the data specified in the source register field and the immediate operand and the result is stored in the destination register field. For example : ADD R_1, # 100, R_2 ; $R_2 \leftarrow R_1 + 100$

Relative to PC addressing : In relative to PC addressing, the instruction usually consists of three fields : opcode field, condition field and address field. Opcode field specifies the operation. The condition field specifies one of many possible branch conditions and address field specifies the signed offset which is to be added to the contents of PC to calculate new address when branch condition is satisfied. For example, JMP COND, $R_1(R_2)$; PC $\leftarrow R_1 + R_2$ if condition is satisfied.

Review Question

1. Explain the addressing modes of RISC architecture.

6.5 RISC Evaluations

In this section we discuss the advantages of RISC architecture from a number of points of view such as

- VLSI realization
- Computing speed
- Design cost and reliability
- HLL support

6.5.1 RISC and VLSI Realization

We have seen that, the RISC has relatively few instructions, few addressing modes, and few instruction formats. Due to this, for RISC machines, relatively small and simple (compared to CISC) decoding and executing hardware subsystem of the CPU is required. This gives following results when we realize computing system by VLSI chip.

1. The control unit for RISC machines require considerably reduced chip area; below 10% of chip area. In general, the control unit for CISC machines might take up over 50% the chip area. As a result, on a RISC VLSI chip, there is more area available for other features such as cache, FPU, part of main memory, memory management unit, I/O ports and so on.

2. Due to reduced control area, RISC designer can provide large number of CPU registers on the chip. This in turn increases the throughput.
3. Due to reduced control area on a VLSI chip and providing numerous identical registers in the additional space, we can increase the regularization factor of the chip. The regularization factor is defined as the total number of devices on chip, excluding ROMs, divided by the number of drawn devices (such as registers, ALUs, counters, and other subsystems. As regularization factor increases the VLSI design cost decreases.
4. The GaAs VLSI chip realization technology is currently limited to a relatively low density compared to CMOS VLSI chip realization technology. Therefore, since a RISC reduces the control area, it represents an another approach for GaAs, single chip, CPU realization.

6.5.2 Computing Speed

In RISC machines, the uniformity of instruction size and duration of execution, wait or hold periods in the pipeline are reduced to a minimum. The design approach of RISC machines is also suitable for efficient handling of pipeline. The above two factors in the RISC machines increase the computing speed. There are also other factors which also increase the computing speed of RISC machines which are discussed below :

1. Shorter propagation delays due to fewer gates in the simpler and smaller control unit.
2. Faster decoding operation due to simpler and smaller decoding system.
3. Reduction in number of memory accesses reduced due to large number of CPU registers.
4. The use of delayed branch technique prevents the flushing of prefetched instructions in case of a successful branch. Thus increases the computing speed.

6.5.3 Design Cost and Reliability Considerations

A relatively simple and small control unit in a RISC CPU usually gives the following design cost and reliability benefits :

1. Design cost is less due to less time required to design simple and small control unit of RISC machines.
2. A shorter design time would reduce the probability that the end product will be obsolete by the time the design is completed.
3. Less design errors due to simple and small control unit. This increases reliability.

4. Due to uniformity in the instruction formats, instruction will not cross word boundaries and an instruction can not wind up on two separate pages in a virtual memory. This makes the design of a virtual memory management subsystem simple.

6.5.4 HLL Support

Like CISC processors, the RISC machines also offer some features that directly support common HLL operations and simplify the design of certain HLL compilers. These features are explained below :

1. Number of instructions in RISC machines are less. Because of this, there is only a single choice for the compiler to realise certain operations. This makes that part of the compiler short and simple.
2. The availability of large number of CPU registers permits a more efficient code optimization in a compiler by maximizing the number of faster register-to-register operations and minimizing the number of slower memory accesses.
3. The register windows in the RISC machines allow fast parameter passing between procedures. This constitutes a direct support of HLL handling of subroutines and procedures.
4. Since RISC instruction set reduces burden on the compiler writer. This results less time to develop it and with less cost.

6.5.5 Shortcomings of RISC Architecture

A number of functions performed on CISC machine by a single instruction, will need two, three or more instructions on a RISC machine. This makes RISC code longer and requires more memory for storage. This also increases instruction traffic between the memory and the CPU. In general, a RISC program is about 30 percent longer than a CISC program.

Review Questions

1. *Comment on VLSI realization of RISC processor.*
2. *Describe the factors which increase computing speed in RISC machine.*
3. *Explain how design cost decreases and reliability increases in RISC machines.*
4. *Comment on HLL support provided by RISC machines.*

6.6 On-chip Register File Versus Cache Evaluation

RISC systems have a large on-chip register file (CPU register file). A large on-chip register file has some advantages and disadvantages. Let us discuss them.

Advantages of large on-chip (CPU) Register File

- By keeping the operands, intermediate results and data which is needed later in the program in the register file the overall data traffic between the memory and the CPU is reduced increasing the speed of operation.
- Large register set allows efficient parameter passing between procedures. With a small register set, most of the parameters have to be stored in memory and fetched from it whenever needed, increasing the CPU memory traffic.
- Each task has a certain data associated with it known as the **task context** or the **task state.** In a multitasking environment processor is often switched between different tasks. At the time of task switching it is necessary to store the task context of the interrupted task so that it can be performed later in the program sequence. If the CPU register file is large and the task context is limited to a finite subset of the CPU registers, the task context can be saved in another subset of the CPU register file, saving on CPU-memory traffic. This is not possible with small CPU register file.
- A large CPU-register file allows to implement data structures like stack and queue on the CPU-chip itself.
- A large number of identical registers on the CPU chip increases its regularization factor and thus reduces the VLSI design and manufacturing cost.

Disadvantage of a large on-chip (CPU) Register File

- A large on-chip register file has more complicated decoding system, increasing the access time to any of its registers.
- Large on-chip register file which uses window pointers also tends to increase the register address decoding time.
- Window management policies for large on-chip register file may also complicate the CPU logic, raising the cost and slowing down the operation.
- Large CPU register file takes more chip space.
- Advanced compiler technology makes efficient use of relatively small register files (16 to 32 registers).
- If all CPU registers are required to save on a context switch, a large register file takes more time to store or retrieve.

Alternative Approach : Use of Cache

An alternative approach to a large register file is the use of a cache (cache is a small part of fast memory consists of static RAM). This approach has some advantages. The Table 6.6.1 gives the comparison between the merits of a cache and a register file approaches.

No.	Cache	CPU Register File
1.	Cache addresses are actually memory addresses (usually 32-bit). They take longer to decode.	Register addresses are short addresses (For example : 7 bits for 128 registers).
2.	If cache is on the CPU chip it has less access time, but still more than access time for CPU registers. If cache is not on the CPU chip access time is large.	It takes less access time.
3.	It has to be tens of kbytes to be effective	It has to be 128 registers (512 bytes) to give significant effect on the performance.
4.	It has slower access due to time elapses in effective address (EA) calculation and virtual to physical address translation.	It has faster access.
5.	The information loaded in the cache is based on prefetch and replacement policies.	Any information can be loaded at any time in the CPU register file.
6.	The information loaded in units of lines (blocks).	The information can be loaded individually to each register.
7.	It is usually not accessible to the user.	It is fully accessible by the user.
8.	Possibility of cache miss.	No question of miss.

Table 6.6.1 On-chip cache versus register file

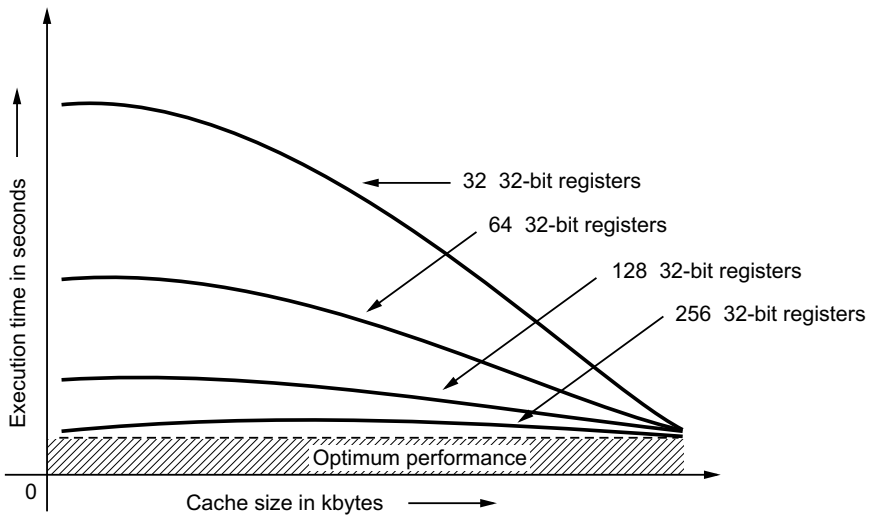


Fig. 6.6.1 Register/cache ratio versus performance

The Fig. 6.6.1 shows the effect of cache and on-chip register file on performance in the form of graph. The horizontal axis represents the on-chip data cache and the vertical axis represents computing time. The four curves represent four different sizes of the register file : 32, 64, 128 and 256 registers. The Fig. 6.6.1 shows that doubling the register numbers from 32 to 64 improves the performance considerably.

Review Questions

1. *Give the disadvantages of having large on-chip CPU register file.*
2. *Give the comparison between the on-chip cache and CPU register file.*

6.7 Overview of RISC Development and Current Systems

The sun SPARC, Intel i860, Motorola M88100 and AMD 29000 are the examples of RISC architecture. There are on-chip floating point units in i860 and M88100 whereas off-chip floating point units in SPARC and AMD. The instructions used by all of these processors are 32-bit. The instruction set of each processor consists of 51 to 124 basic instructions. Let us study the currently popular RISC systems.

6.7.1 SPARC

'SPARC' stands for scalable processor architecture. In the previous section we have seen the concept of register window. A number of register windows in SPARC architecture determines its scalability.

There are different chips of SPARC designed by different manufacturers. Different number of register windows are used by different chips. The following Table 6.7.1 summarizes the details of various SPARC chips.

SPARC Chip	Technology	Clock rate (MHz)	Claimed VAX MIPS
Cypress CY7C601 IU	0.8 - µm CMOS IV, 207 pins	33	24
Fujitsu MB 86901 IU	1.2 - µm CMOS, 179 pins.	25	15
LSI Logic L64811	1.0 - µm HCMOS, 179 pins.	33	20

TI 8846	0.8 - µm CMOS	33	24
BIT IU B-3100	ECL family.	80	50

Table 6.7.1 SPARC implementations by licensed manufacturers

We have chosen SUN SPARC CY7C601 for studying specifications and architectural details.

Specification of SUN SPARC CY7C601

Let us first see the specifications of SUN SPARC CY7C601

1. Instruction set is of 69 instructions.
2. Instructions are of 32-bit format.
3. 7-data types.
4. 4-stage instruction pipeline.
5. 32-bit RISC/IU.
6. 136 registers divided into 8 windows.
7. Off-chip cache.
8. MMU on CY7C604 with 64-entry TLB.
9. Off-chip FPU (Floating Point Unit) on CY7C602.
10. There are 32 registers and 64-bit pipeline in FPU.
11. Concurrent IU and FPU operations.
12. 0.8 - µm CMOS IV technology is used.
13. Clock rate is 33 MHz.
14. 207 pins package (in 1989).
15. 24 MIPS for 33 MHz version.
    50 MIPS for 80 MHz ECL version.
16. Upto 32 register windows can be built.

Architecture of SUN SPARC

Fig. 6.7.1 (a) shows the architecture of the Cypress CY7C601 SPARC processor and Fig. 6.7.1 (b) shows the architecture of CY7C602 FPU.

There is a set of thirty-two 32-bit IU registers. The SPARC runs each procedure with this set of registers. Out of 32, eight registers are global registers and twenty-four are window registers. Global registers are shared by all procedures and window registers are associated with only each procedure.

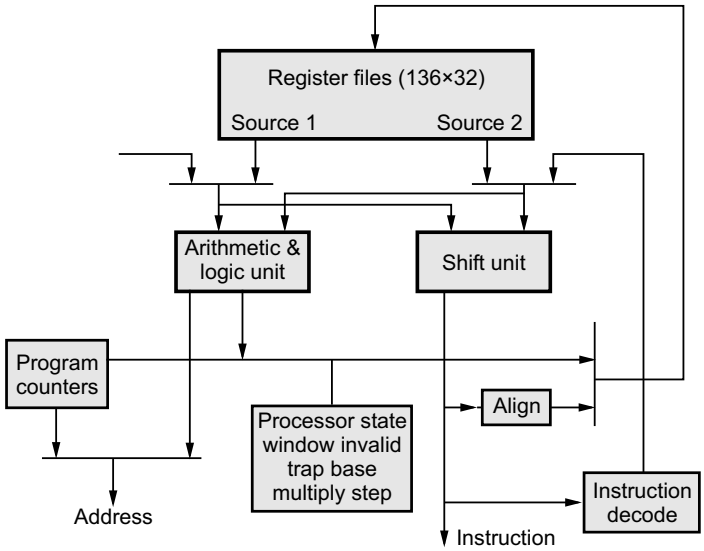


Fig. 6.7.1 (a) Cypress CY7C601 SPARC processor

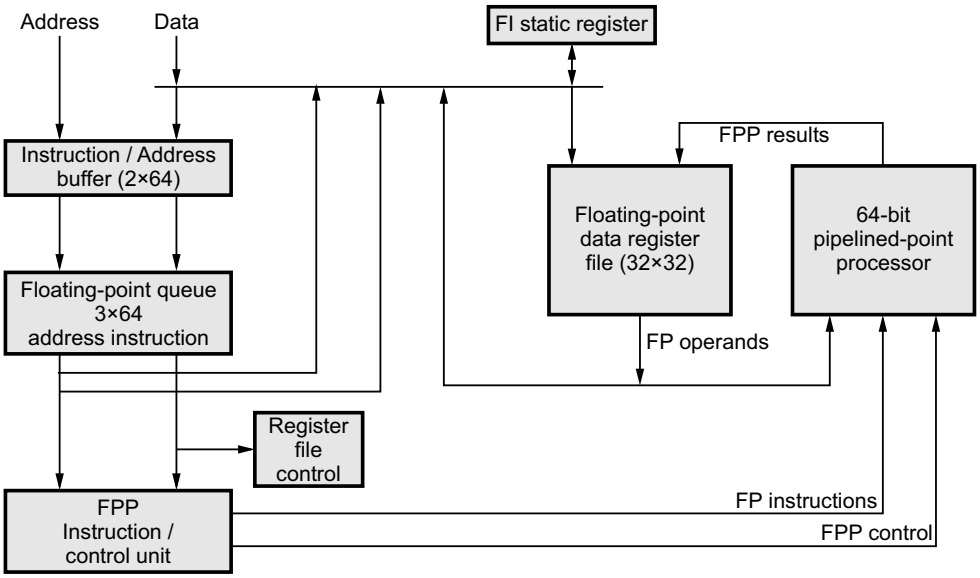


Fig. 6.7.1 (b) Cypress CY7C602 floating-point unit

Register Windows

A unique feature contributing to the high performance of the SPARC design is its overlapping register windows. The concept of overlapping windows is illustrated in Fig. 6.7.2.

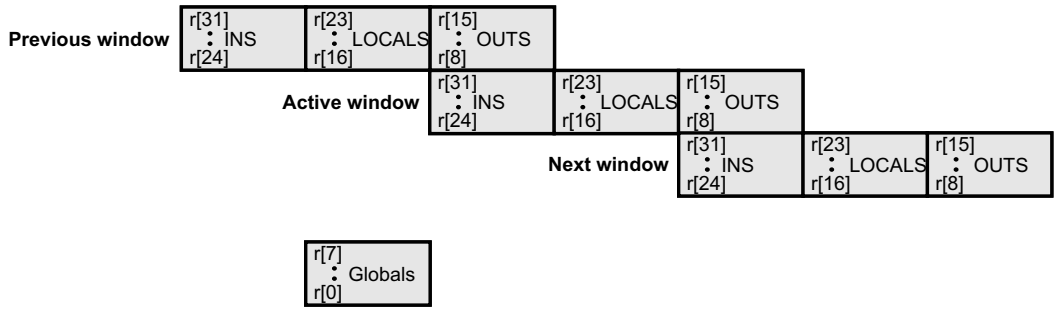


Fig. 6.7.2 (a) Three overlapping register windows and the global register

As shown in Fig. 6.7.2, there are eight overlapping windows and eight globals. Eight overlapping windows are formed with 64 local registers and 64 overlapped registers. Thus there are totally 136 registers.

As shown in Fig. 6.7.2, each register window is divided into three sections. Each section is of eight registers. These sections are labeled as Ins, Locals and Outs. The Ins and Outs are shared among procedures. The locals are only locally addressable by each procedure. The r8 to r15 are the Outs registers of the calling procedure. The calling procedure passes parameters to the called procedure via its Outs registers. These Outs registers are the Ins registers of the called procedure. The active window is the window of the currently running procedure. This is pointed to by a current window pointer. i.e. CWP. The invalid window is indicated by a window invalid mask. The trap base register is used as a pointer to a trap handler. For a multiple step instruction, a special register is used to create a 64-bit product. The procedures can also be called without changing the window. For a function call, the register windows rotate counterclockwise and for a return from a function call, they rotate clockwise.

Register windows have several advantages over a fixed set of registers as listed below.

1. By acting like a cache, they reduce the number of load and store instructions issued by a compiler. A consequence is a decrease in the chip/cache and cache/memory operand bandwidth. i.e. fewer loads and stores and fewer data cache misses. This benefits chips with multi-cycle load or store instructions by executing fewer of them and tightly coupled, cache-consistent multiprocessors by reducing the memory bus traffic.
2. Windows perform well in incremental complication environments such as LISP and in object-oriented languages such as Smalltalk.
3. A special register is used to create a 64-bit product in multiple step instructions. Procedures can also be called without changing the window.

4. The overlapping windows can significantly save the time required for interprocedure communications. This results in much faster context switching among cooperative procedures.

As shown in Fig. 6.7.3, FPU features 32 single-precision (32-bit) or 16 double-precision (64-bit) floating point registers. Out of 69 instructions, fourteen instructions from the SPARC instruction set are for floating point operations.

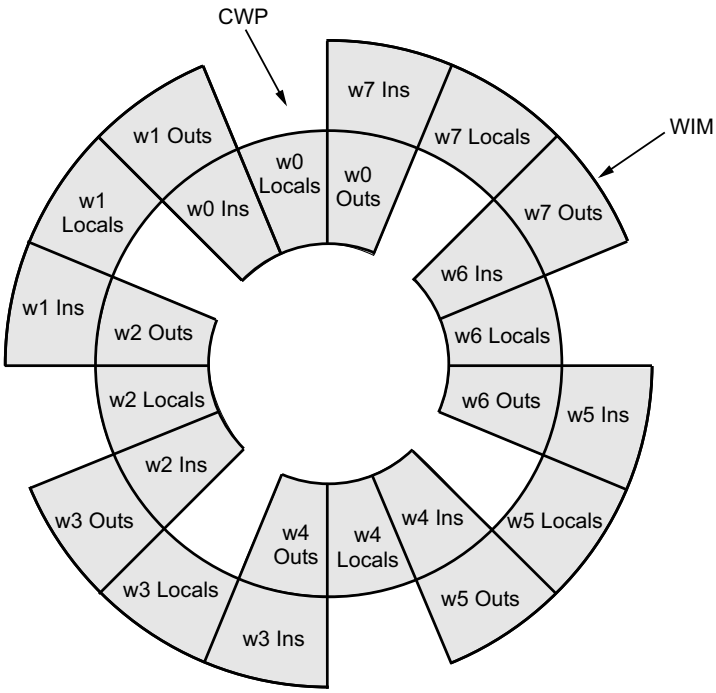


Fig. 6.7.3 Eight register windows forming a circular stack

6.7.2 The Intel i860 Processor Architecture

This is a 64-bit RISC processor fabricated on a single chip. Its specifications are listed below.

1. Its instruction set consists of 82 instructions. (42 RISC integer, 24 floating-point, 10 graphics and 6 assembler pseudo operations).
2. 32-bit instruction format.
3. Four addressing modes.
4. 32-bit RISC core.
5. There are 32 registers.
6. 4 kB code, 8-kB data, on-chip MMU, paging with 4 kB/page.

7. On-chip 64-bit floating point (FP) multiplier and FP adder with 32 FP registers, 3-D graphics unit.
8. It allows duel instructions and dual FP operations.
9. Operating frequency (clock rate) is 40 MHz.
10. 1-μm CHMOS IV technology is used.
11. 168-pins package.
12. 10 MIPS and 60 Mflops for 40 MHz.
13. 1M transistors

    (2.5 M transistory - i860/XP).

Fig. 6.7.4 shows the functional block diagram of i860 processor. As shown in Fig. 6.7.4, there are nine major functional units. The data path width is not same throughout the block diagram. The bus-width varies from 32 to 128 bits as given below.

1. All external or internal address buses are 32-bit wide.
2. The external data path or internal data bus is 64-bits wide.
3. The data cache is two-way set associative of 8 kbytes. It transfers 128 bits per clock cycle. so cache data bus is 128 bits wide.
4. The bus control unit transfers 64-bits data between the chip and the outside world. So external data bus connected to bus control unit is 64-bits wide.
5. The internal RISC integer ALU is 32-bits wide. So here data bus width is 32-bit.
6. The instruction cache is a two-way set-associative memory of 8 kbytes. It transfers 64-bits per clock cycle.

The functions of different blocks in the block diagram are explained below.

- **Instruction Cache :** It is 4 kbytes memory organized as a two-way set-associative with 32 bytes per cache block. It transfers 64 bits per clock cycle. This is equivalent to 320 Mbytes/s at 40 MHz.
- **Data Cache :** It is also a two-way set-associative memory of 8 kbytes. It transfers 128 bits per clock cycle. This is equivalent to 640 Mbytes/s at 40 MHz. A write-back policy is used here. It needed, cacheing can be inhibited by software.
- **Memory Management Unit (MMU) :** It implements protected 4 kbyte paged virtual memory of 2^{32} bytes via a TLB.
- **Bus Control Unit :** The 64-bit data transfer between the chip and the outside world is coordinated by the bus control unit.
- **RISC Integer Unit :** It performs the functions of executing load, store, integer, bit and control instructions. It also fetches instructions for the floating-point control unit.

- **Floating Point Units :** There are two floating point units. These are, the multiplier unit and the adder unit. These two units can be used separately or simultaneously. This is done under the coordination of the floating point control unit.

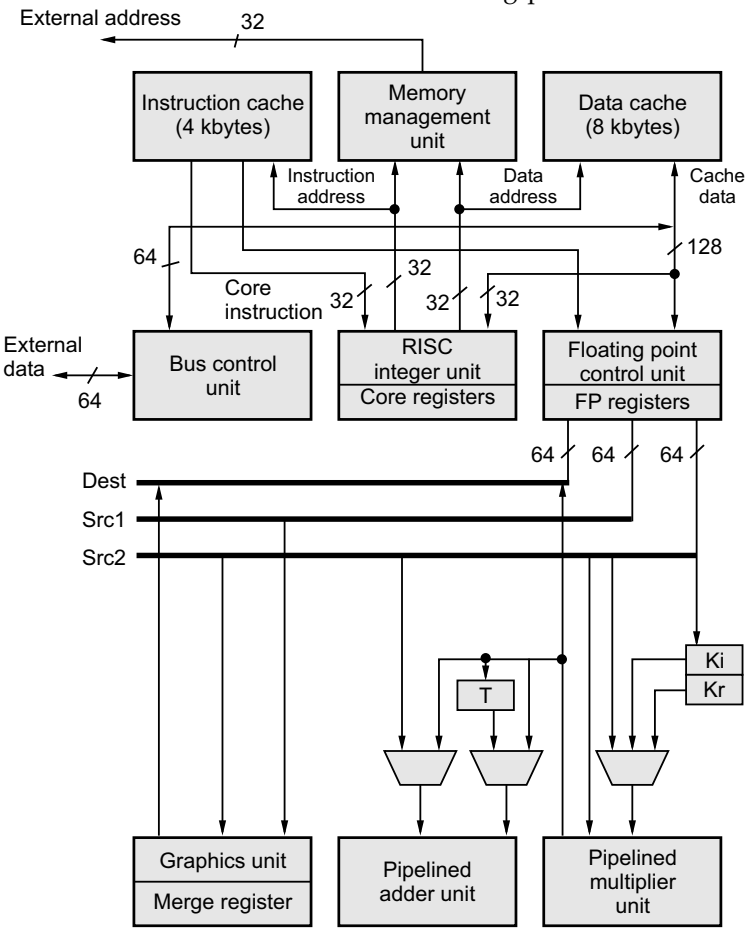


Fig. 6.7.4

There are special dual-operation floating point instructions. For example-add and multiply. This uses both, the multiplier and adder units in parallel. Also, both the integer unit and the floating-point unit can execute concurrently. Thus i860 processor can execute two instructions, one integer and one floating-point, simultaneously. By considering this point, the i860 is a superscaler RISC processor.

- **Graphics Unit :** This unit executes integer operations corresponding to 8-, 16-, or 32-bit pixel data types. This unit supports three-dimensional drawing in a graphics frame buffer. The color intensity, shading, hidden surface elimination, such operations can be performed in this unit. There is one register, called as 'merge register'. It is used only by vector integer instructions. The results from multiple addition operations are merged (accumulated) in this register.

6.7.3 RISC Processor-Motorola 88000

The M88000 is a first RISC processor from Motorola. The first members of the M88000 family are the MC 88100 CPU and the MC 88200 cache MMU chips. They were followed by the new generation RISC MC 88110.

Architecture

The Fig. 6.7.5 (a) shows the block diagram of MC88110 processor from M88000 family. It is a two issue superscaler system with 10 on-chip execution units. The architecture defines a major opcode subset denoted as a special function unit (SFU). All floating point instructions are encoded in SFU1 and all graphics instructions in SFU2, as shown in the Fig. 6.7.5 (b).

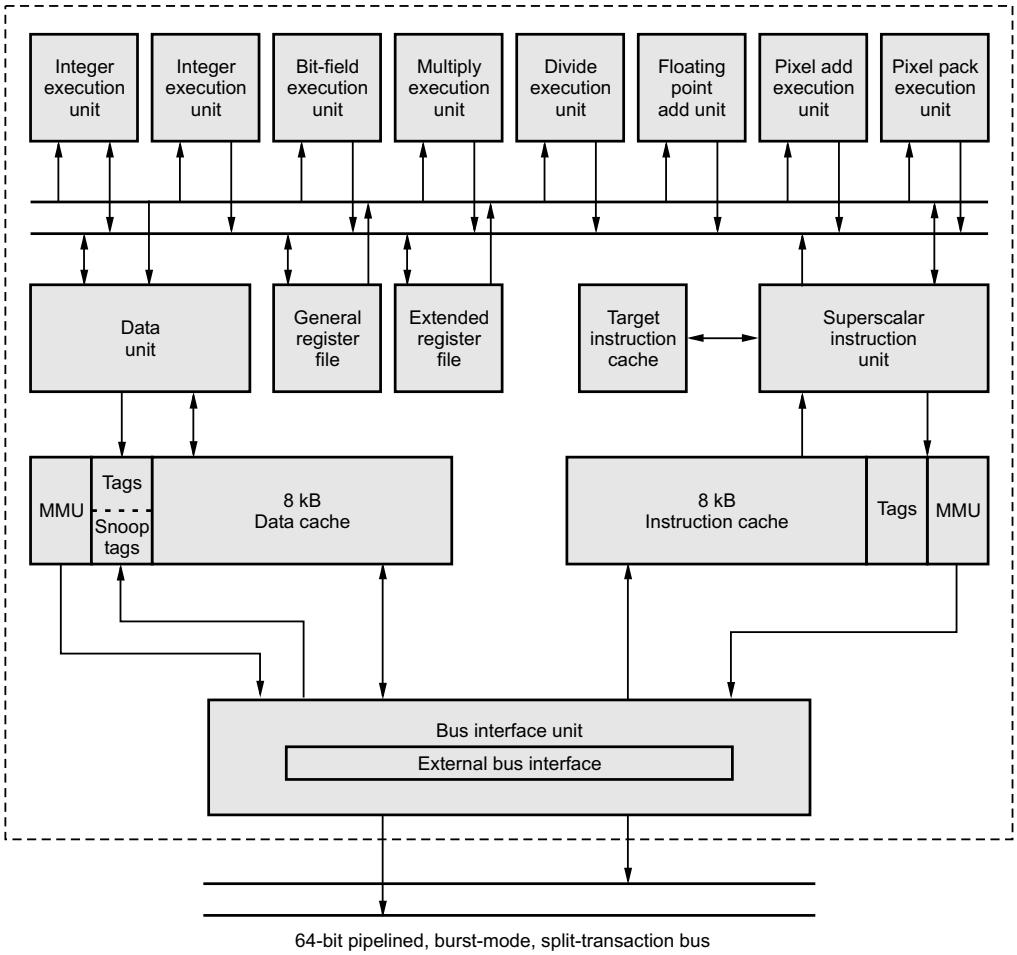


Fig. 6.7.5 (a)

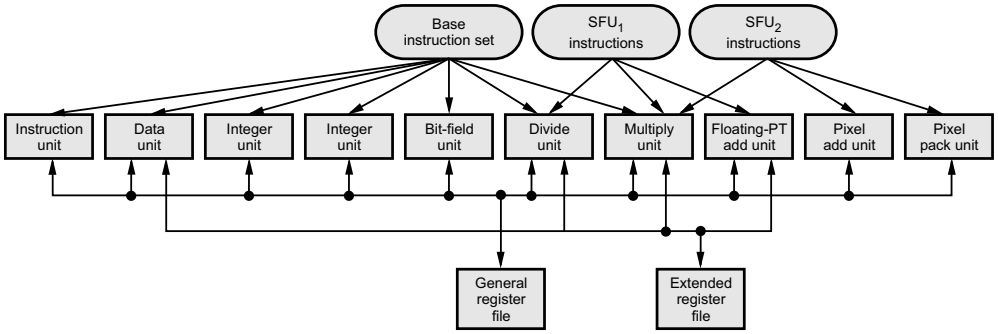


Fig. 6.7.5 (b)

The ten execution units permits simultaneous execution of a large number of combinations of simultaneously fetched instructions. Even if both instructions are integer arithmetic, or both are graphics instructions, there are two identical execution units to execute them in parallel. This is the biggest advantage of the superscalar system. There is also a fully associative, 32-entry logically addressed, target instruction cache (TIC). This cache eliminates much of the delay involved with changes in instruction flow.

Register Organization

The Fig. 6.7.6 shows the register organization for M88000 family processors. It consists of following registers:

- General register file (GRF)
- Extended register file (XRF)
- Floating-point control register (FPCR), 32-bit
- Floating-point status register (FPSR), 32-bit
- Control registers CR_0 to CR_{51}, 32-bit
- Floating-point exception cause register (FPECR), FCRO, 32-bit.

Data Types

The M88000 family processors support following data types for operands :

1.	**Integer**	:	Byte - 8 bits
			Halfword - 16 bits
			Word - 32 bits
			Double word - 64 bits
2.	**Bit field**	:	1 to 32 bits in a 32-bit register
3.	**Floating point**	:	Single precision - 32 bits
			Double precision - 64 bits
			Double extended precision - 80 bits

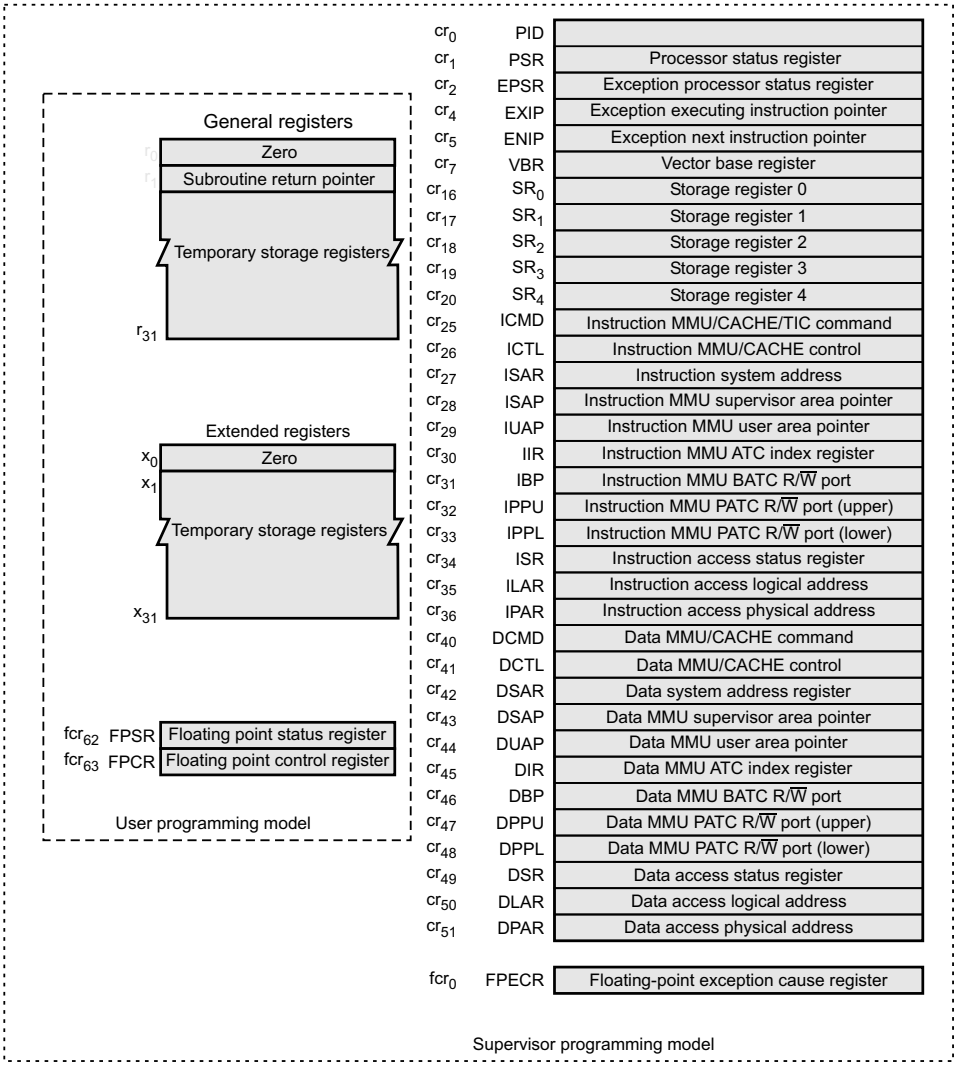

General registers
Zero
Subroutine return pointer
Temporary storage registers
r31
Extended registers
x0
Zero
x1
Temporary storage registers
x31
fcr62 FPSR Floating point status register
fcr63 FPCR Floating point control register
User programming model
cr0 PID
cr1 PSR Processor status register
cr2 EPSR Exception processor status register
cr4 EXIP Exception executing instruction pointer
cr5 ENIP Exception next instruction pointer
cr7 VBR Vector base register
cr16 SR0 Storage register 0
cr17 SR1 Storage register 1
cr18 SR2 Storage register 2
cr19 SR3 Storage register 3
cr20 SR4 Storage register 4
cr25 ICMD Instruction MMU/CACHE/TIC command
cr26 ICTL Instruction MMU/CACHE control
cr27 ISAR Instruction system address
cr28 ISAP Instruction MMU supervisor area pointer
cr29 IUAP Instruction MMU user area pointer
cr30 IIR Instruction MMU ATC index register
cr31 IBP Instruction MMU BATC R/W port
cr32 IPPU Instruction MMU PATC R/W port (upper)
cr33 IPPL Instruction MMU PATC R/W port (lower)
cr34 ISR Instruction access status register
cr35 ILAR Instruction access logical address
cr36 IPAR Instruction access physical address
cr40 DCMD Data MMU/CACHE command
cr41 DCTL Data MMU/CACHE control
cr42 DSAR Data system address register
cr43 DSAP Data MMU supervisor area pointer
cr44 DUAP Data MMU user area pointer
cr45 DIR Data MMU ATC index register
cr46 DBP Data MMU BATC R/W port
cr47 DPPU Data MMU PATC R/W port (upper)
cr48 DPPL Data MMU PATC R/W port (lower)
cr49 DSR Data access status register
cr50 DLAR Data access logical address
cr51 DPAR Data access physical address
fcr0 FPECR Floating-point exception cause register
Supervisor programming model

Fig. 6.7.6 Register organization for M88000

4. Graphics : 32-bit packed nibble, bytes, and half-words

: 64-bit packed nibble, bytes, halfwords, and words

For each instruction, the data type of operand is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Bit fields are defined by width and offset values given in the instruction or in a source register specified by the instruction.

Addressing Modes

The M88000 family processors support following addressing modes :

- Register
- Immediate
- Register indirect with immediate displacement
- Register indirect with index
- Register indirect with scaled index
- Register indirect
- Register indirect with 9-bit vector table index
- PC-relative with 16-bit displacement
- PC-relative with 26-bit displacement

Instruction Format

The Fig. 6.7.7 shows the instruction formats for M88000 family. The M88000 family processors have 32-bit instruction. For all instructions the first six bits represents the

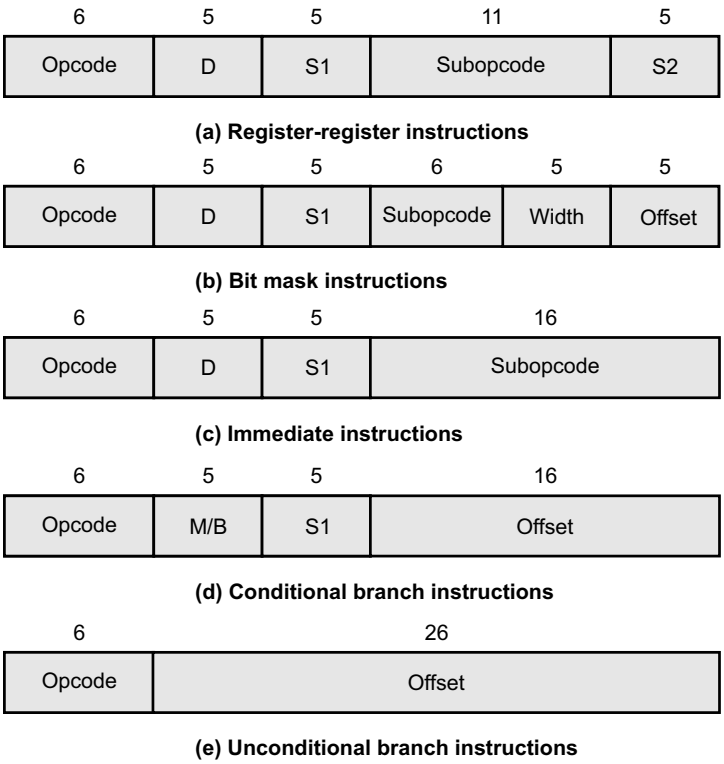


Fig. 6.7.7 Instruction formats for M88000 family

opcode. Some instructions require subopcode, which is specified by the subopcode field. Fields S_1 and S_2 specifies the two source operands and field D specifies the destination operand.

Instruction Set

The Table 6.7.2 lists the instruction set for the 88000.

Integer Arithmetic Instructions	
Mnemonic	**Description**
add	Signed Add
addu	Unsigned Add
cmo	Integer Compare
divs	Signed Divide
divu	Unsigned Divide
muls	Signed Multiply
mulu	Unsigned Multiply
sub	Signed Subtract
subu	Unsigned Subtract

Bit-Field Instructions	
Mnemonic	**Description**
clr	Clear Bit Field
ext	Extract Bit Field
ext u	Unsigned Extract Bit Field
ff0	Find Fist Bit Clear
ff1	Find Fist Bit Set
mak	Make Bit Field
rot	Rotate Register
set	Set Bit Field

Logical Instructions	
Mnemonic	**Description**
and	And
mask	Logical Mask Immediate
or	Or
xor	Exclusive Or

Graphics Instructions	
Mnemonic	**Description**
padd	Pixel Add
padds	Pixel Add and Saturate
pcmp	Pixel Compare
pmul	Pixel Multiply
ppack	Pixel Truncate, Insert, and Pack
prot	Pixel Rotate Left
psub	Pixel Subtract
psubs	Pixel Subtract and Saturate
punpk	Pixel Unpack

Flow Control Instructions	
Mnemonic	**Description**
bb0	Branch on Bit Clear
bb1	Branch on Bit Set
bcnd	Condition Branch
br	Unconditional Branch

bsr	Branch to Subroutine
illop	Illegal Operation
jmp	Unconditional Jump
jar	Jump to Subroutine
rte	Return from Exception
tb0	Trap on Bit Clear
tb1	Trap on Bit Set
tbnd	Trap on Bounds Check
tcnd	Conditional Trap

Load/Store/Exchange Instructions	
Mnemonic	**Description**
ld	Load Register from Memory
lda	Load Address
ldcr	Load from Control Register
st	Store Register to Memory
stcr	Store to Control Register
xcr	Exchange Control Register
xmem	Exchange Register with Memory

Floating-Point Instructions	
Mnemonic	**Description**
fadd	Floating-Point Add
fcmp	Floating-Point Compare
fcmpu	Unordered Floating-Point Compare
fcvt	Convert Floating-Point Precision

fdiv	Floating-Point Divide
fldcr	Load from Floating-Point Control Register
flt	Convert Integer to Floating-Point
fmul	Floating-Point Multiply
fsqrt	Floating-Point Square Root
fstcr	Store to Floating-Point Control Register
fsub	Floating-Point Subtract
fxcr	Exchange Floating-Point Control Register
int	Round Floating-Point to Integer
mov	Register-To-Register Move
nint	Round Floating-Point to Nearest Integer
trnc	Truncate Floating-Point to Integer

Table 6.7.2 Motorola 88000 instruction set

Review Questions

1. *Draw and explain the graph showing the register/cache ratio versus performance.*
2. *Write a short note on overlapping register window.*
3. *Explain typical block diagram of any one RISC processor.*

6.8 Superscalar Processor

May-11

Microprocessors such as the PowerPC reach performance levels greater than one instruction per cycle by fetching, decoding and executing several instructions concurrently. This mode of operation is called **superscaler**. A processor capable of parallel instruction execution and having performance level greater than one instruction per cycle is known as **superscaler processor**. A superscaler processor has multiple execution units (E-units), each of which is usually pipelined, so that they constitute a set of independent instruction pipelines. Its program control unit (PCU) is capable of fetching and decoding several instructions concurrently. It can issue up to k instructions simultaneously to the various E-units. Where k is called **instruction issue degree**. It can be six or more using current technology.

To fetch, decode and execute several instruction concurrently, the superscaler operation requires processor to detect and exploit instruction-level parallelism hidden in the programs it executes. It has to handle following important issues :

- **Instruction type :** A floating-point add instruction has to be issued to a floating point E-unit and not to an integer E-unit.
- **E-unit availability :** An instruction can be issued to a pipelined E-unit only if no collision will result, as determined by the pipeline's reservation table.
- **True data dependency**
- **Procedural dependency**
- **Resource conflicts**
- **Output dependency**
- **Antidependency**

True Data Dependency

Let us see the following instructions :

Sub r_1, r_2 ; Subtract the contents of r_2 from r_1 and store the result in r_1

Move r_3, r_1 ; Load register r_3 with the contents of r_1

In the above instruction sequence, the second instruction can be fetched and decoded, but cannot be executed until the first instruction executes. The reason is that the second instruction requires the data produced by the first instruction. This situation is referred to as a **true data dependency**. This limits the performance of any type of pipeline.

Procedural Dependency

We have seen earlier that the branch instructions complicate the pipeline operation. There are branch prediction techniques, but they cannot improve performance hundred percent. In branch prediction technique, **taken or not taken**, we cannot execute instruction until branch is executed. This situation is referred to as **procedural dependency**.

Resource Conflict

While executing number of instructions in parallel, the situation may occur that two or more instructions require same resource at the same time. For example, two or more instructions may require references to memories, caches, buses register-file ports, and functional units at the same time. This situation is referred to as **resource conflict**.

Output Dependency

Let us see the following instructions.

R3 = R3 + R5 ← I1
R4 = R3 + 1 ← I2
R3 = R5 + 1 ← I3
R6 = R3 + R4 ← I4

In the above instruction sequence we cannot execute instruction I2 before I1 because of the true data dependency and for similar reasons we also cannot execute instruction I4 before I3. But there is one more dependency which exists between instruction I3 and instruction I1. We cannot execute instruction I3 before instruction I1, because if I3 executes prior to completion of I1, then wrong value of the contents of R3 will be fetched for the execution of instructions I1 and I4. This situation is referred to as **output dependency**.

Antidependency

Let us see the following instructions

R3 = R3 + R5 ← I1
R4 = R3 + 1 ← I2
R3 = R5 + 1 ← I3
R6 = R3 + R4 ← I4

In the above instruction sequence instruction I3 cannot be completed its execution before instruction I2 begins execution and had fetched its operands. This is so because instruction I3 updates register R3, which is a source operand for instruction I2. This situation is referred to as **antidependency**. Here, constraints are exactly reverse that of true data dependency.

Control dependencies : To maintain high performance levels, techniques are needed to reduce the impact of branch instructions on pipeline efficiency.

Program order : Instructions must eventually produce results in the order specified by the program being executed. The result may, however, be computed out-of-order internally to improve the CPU's performance.

6.8.1 Instruction Level Parallelism and Machine Parallelism

When instructions in a sequence are independent and can be executed in parallel by overlapping, there is an instruction level parallelism. On the other hand, machine parallelism is a measure of the ability of the processor to take advantage of instruction-level parallelism. Machine parallelism is determined by the number of instructions that can be fetched, decoded and executed at the same time.

Both instruction level and machine parallelism are important factors in enhancing performance. The use of simple and fixed-length instruction set architecture, RISC, enhances instruction level parallelism.

6.8.2 Instruction-Issue Policy

Considering the limitations of parallelism processor must able to identify the next instruction to be fetched, decoded and executed. This process is known as *instruction issue* and each processor has its own *instruction issue policy*.

Superscalar processors use following instruction issue policies.

- In-order issue with in-order completion
- In-order issue with out-of-order completion
- Out-of-order issue with out-of-order completion

In-order Issue with In-order Completion

In this policy instructions are executed in exact order that would be achieved by sequential execution.

Let us consider the instructions with following constraints.

- I1 requires two cycles to execute
- I3 and I4 conflict for function unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for function unit

With above constraints in-order issue with in-order completion policy executes all instructions as shown in the Fig. 6.8.1 (a).

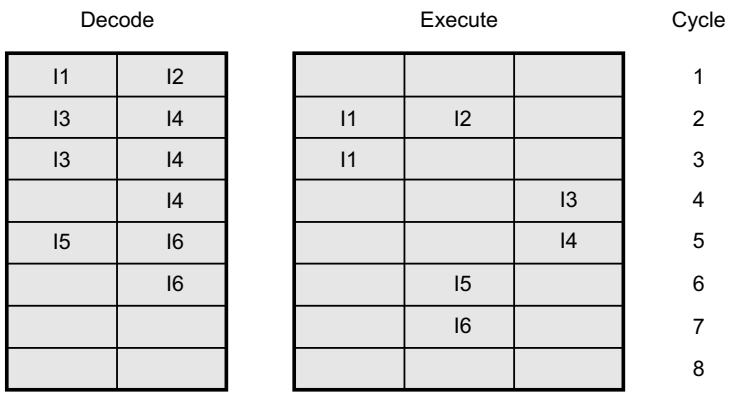


Fig. 6.8.1 (a) In order issue and in order completion

In-order Issue with Out-of-order Completion

If any instruction requires multiple cycles and the next instruction is independent of current instruction then it is allowed to execute before the completion of current instruction. This policy is referred to as in-order issue with out-of-order completion. This policy is used in scalar RISC processors to improve the performance of the instructions that require multiple cycles. Fig. 6.8.1 (b) shows the modified execution sequence due to this policy. As shown in the Fig. 6.8.1 (b) the net result is saving of one cycle.

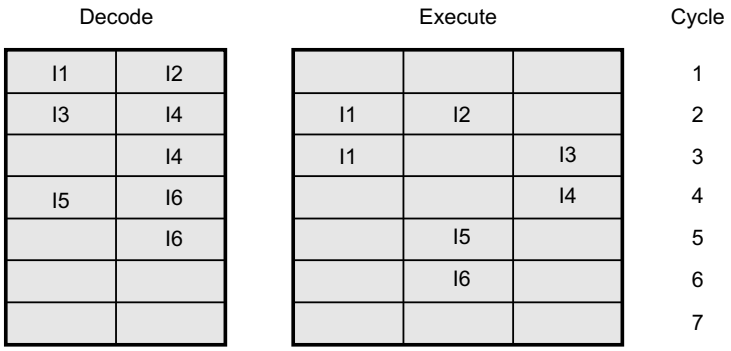


Fig. 6.8.1 (b) In-order issue and out of order completion

Out-of-order Issue with Out-of-order Completion

With in-order issue, the processor decodes the instructions upto the point of a dependency or conflict. But in out-of-order issue, processor decodes the instruction beyond any dependency or conflict, to achieve look ahead capacity, allowing it to identify independent instructions that can be brought into the execute stage. To implement this, processor provides buffer commonly known as *instruction window* where decoded instructions are stored. Fig. 6.8.2 shows the modified execution sequence due to this policy.

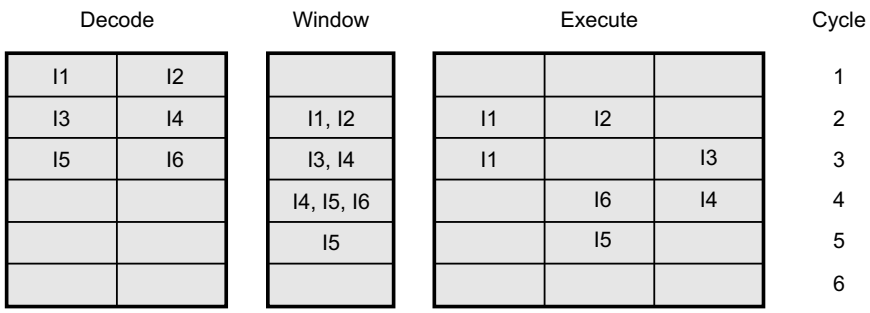


Fig. 6.8.2 Out-of-order issue with out-of-order completion

6.8.3 Register Renaming

The storage conflict can be avoided by duplicating the contents of registers. In this context, the technique is referred to as *register renaming*. To implement register renaming, processor dynamically allocates registers.

6.8.4 Branch Prediction

The branch prediction techniques discussed in section 3.12.4 are used in the super scaler processors to enhance the performance.

6.8.5 Example : PowerPC Processor

The PowerPC architecture, developed jointly by Motorola, IBM and Apple Computer, is based on the POWER architecture implemented by $RS/6000^{TM}$ family of computers. The PowerPC architecture takes advantage of recent technological advances in such areas as process technology, compiler design and reduced instruction set computing (RISC) microprocessor design to provide software compatibility across a diverse family of implementations, primarily single-chip microprocessors, intended for a wide range of systems, including battery-powered personal computers; embedded controllers; high-end scientific and graphics workstations; and multiprocessing, microprocessor-based mainframes.

The PowerPC is a family of processors. The first member of the family is the PowerPC 601 and other include the 603, 604 and 620 processors. All these processors share the PowerPC architecture described in this section. However, they provide different levels of performance because of the difference in their internal cache organizations. (Refer Table 6.8.1)

Processor	Cache size	Bytes/line	Organization
PowerPC 601	1 32-kbyte	32	8-way set associative
PowerPC 603	2 8-kbyte	32	2-way set associative
PowerPC 604	2 16-kbyte	32	4-way set associative
PowerPC 620	2 32-kbyte	64	8-way set associative

Table 6.8.1 PowerPC cache organizations

PowerPC Organization

The Fig. 6.8.3 shows the simplified block diagram of PowerPC 620. It has two caches : instruction cache and data cache. Instead of single ALU, the PowerPC 620 has three integer ALUs to handle integer and logic operations and one floating point ALU to perform floating point operations. It also has an instruction unit whose purpose is to

fetch instructions from the main memory and send them to the integer and floating-point unit for execution.

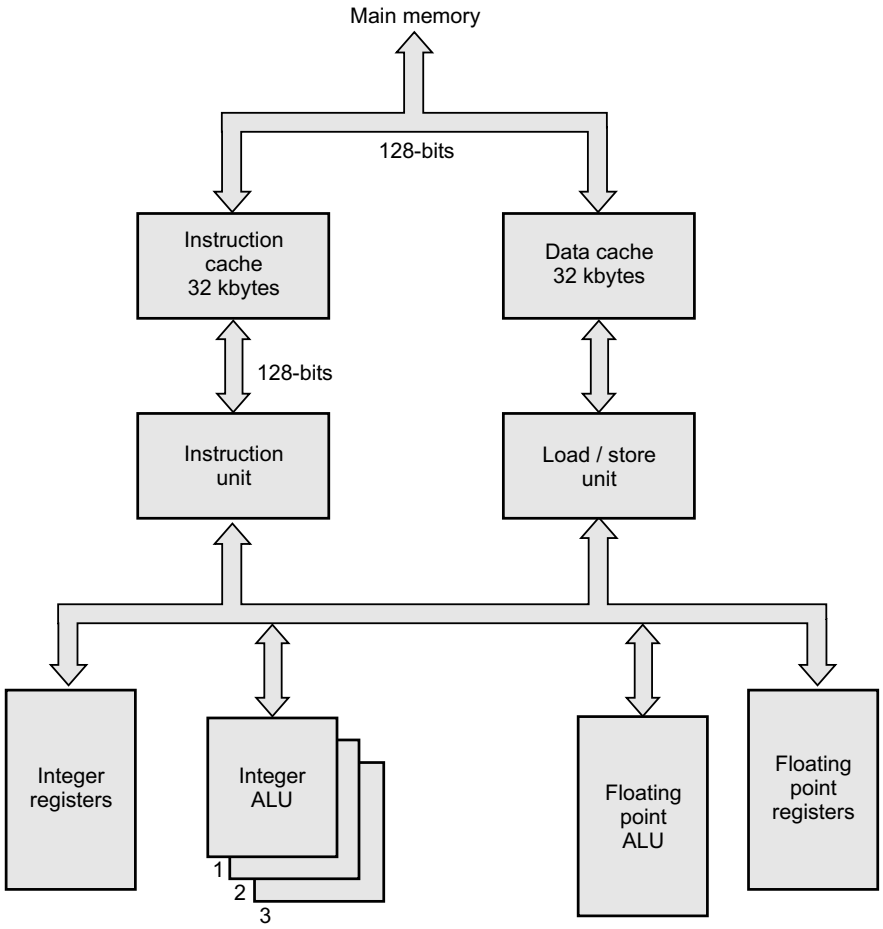


Fig. 6.8.3 Block diagram of PowerPC

PowerPC Register Organization

The Fig. 6.8.4 shows the register organization for PowerPC. There are 32 general purpose registers R_0 to R_{31}, in the integer unit. Each has 64-bits. These registers may be used to load, store and manipulate data operands and may also be used for register indirect addressing. The floating unit has 32 registers, FR_0 to FR_{31}, each 64-bit long. These are used for floating point operations.

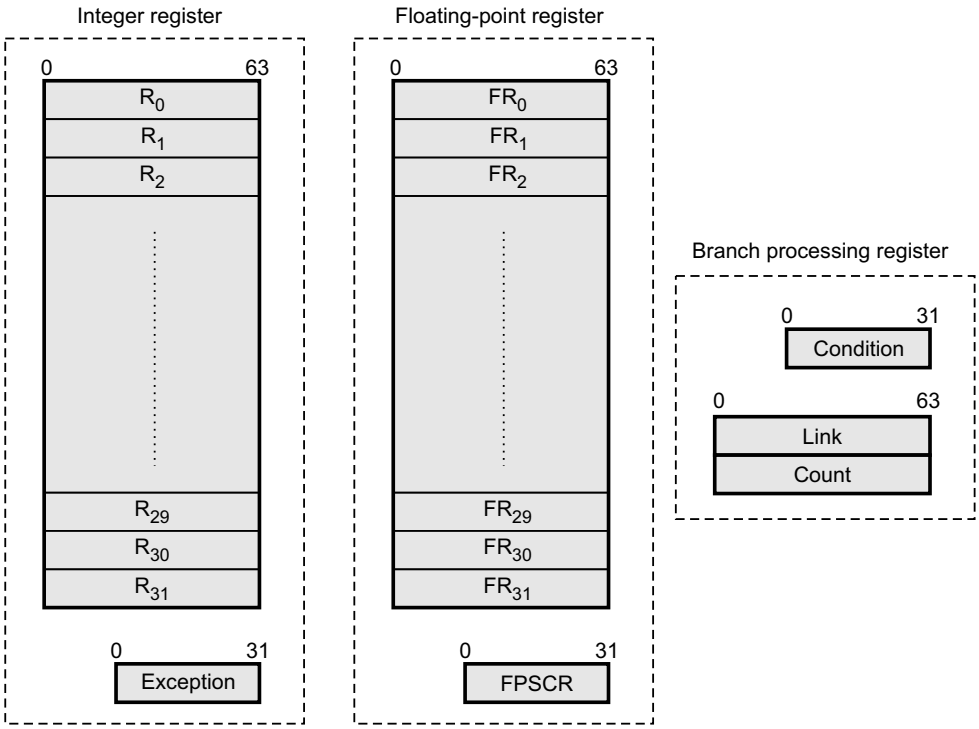
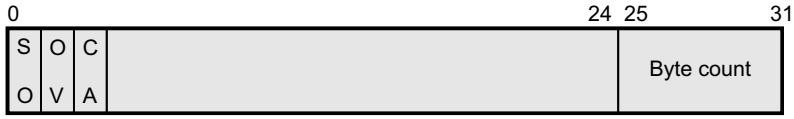


Fig. 6.8.4 PowerPC user accessible registers

The integer unit has one more register called exception register. It is used to indicate exceptions in integer arithmetic operations and to store byte count field. The byte count field is used as an operand for some string instructions. The Fig. 6.8.5 shows the bit pattern for exception register.



Byte count : Specifies number of bytes to be transferred by load/store string indexed instruction
SO : Summary overflow : Set 1 to indicate overflow; remains set until reset by software
OV : Overflow : Set 1 to indicate overflow; reset to 0 by next instruction if there is no overflow
CA : Carry : Set to 1 to indicate carry out of bit 0

Fig. 6.8.5 Exception register

To control the floating operation a 32-bit register is provided in the floating point unit. It is known as floating point status and control register (FPSCR). It contains the control bits as well as status bits. The Table 6.8.2 shows the bit definitions for floating point status and control register.

Bit	Definition
0	Set if any exception occur; remains set until reset by software.
1	Set if any enabled exception has occurred.
2	Set if an invalid operation exception has occurred.
3	Overflow exception
4	Underflow exception
5	Divide by zero exception
6	Inexact exception. Rounded result differs from intermediate result or an overflow occurs with overflow exception disabled.
7 : 12	Invalid operation exception 7 : signaling NaN 8 : $(\infty - \infty)$ 9 : $(\infty + \infty)$ 10 : $(0 + 0)$ 11 : $(\infty \times 0)$ 12 : Comparison involving NaN Note : NaN (Not a Number)
13	Fraction rounded
14	Fraction inexact
15 : 19	Result flags : Five bit code specifies less than, greater than, equal, unordered, quiet NaN, $\pm \infty$, $\pm$ normalized, $\pm$ denormalized, $\pm$ 0.
20	Reserved
21 : 23	Invalid operation exception 21 : Software request 22 : Square root of a negative number 23 : Integer conversion involving a large number, an infinity, or NaN.
24	Invalid operation exception enable
25	Overflow exception enable
26	Underflow exception enable

27	Divide by zero exception enable
28	Inexact exception enable
29	Non-IEEE mode
30-31	Rounding control : Two bit code specifies to nearest, toward 0, toward $\pm \infty$, toward $-\infty$.

Table 6.8.2 Bit definitions for floating point status and control register

The powerPC has three registers in its branch processing unit. Link, count and condition registers.

Link Register : The link register is used for call/return operations. If the LK bit in a conditional branch instruction is set, then the address following the branch instruction is placed in the link register, and it can be used for a later return. It can also be used in a conditional branch instruction for indirect addressing of the target address.

Count Register : It can be used as a counter for iteration loop. The count in the count register is decremented each time it is tested in a conditional branch instruction. This register is also used for indirect addressing of the target address in a branch instruction.

Condition Register : It consists of eight 4-bit condition code field, as shown in the Fig. 6.8.7. Each field of it has number of uses. The Table 6.8.2 gives the interpretation of bits in condition register. As shown in Table 6.8.3 the first four bits (CR_0) are set for all integer arithmetic instructions for which the RC bit is set. This field indicates whether the result of the operation is positive, negative, or zero. The fourth bit is a replica of the summary overflow bit from the exception register. Similarly the next field (CR_1) is set for floating point arithmetic instructions for which the RC bit is set. In this case, the four bits are set according to the first bits of FPSCR register (Refer Table 6.8.3). The remaining eight condition fields (CR_0 through CR_7) can be used with a compare instruction; in each case, the identity of the field is specified in the instruction itself. The first three bits of the field for both fixed-point and floating point compare indicate whether the first operand is less than, greater than, or equal to the second operand. The fourth bit is the summary overflow bit for a fixed-point compare, and an unordered indicator for a floating-point compare.

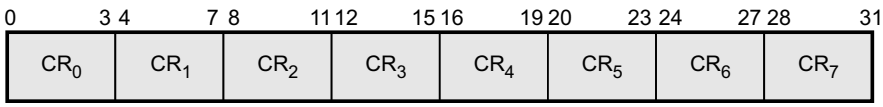


Fig. 6.8.6 Condition register

Bit position in the field	CR_0 (Integer instruction with RC = 1)	CR_1 (Floating point instruction with RC = 1)	CR_2 - CR_7	
			Fixed point compare instruction	Floating point compare instruction
0	Result < 0	Exception summary	$OP_1 < OP_2$	$OP_1 < OP_2$
1	Result > 0	Enabled exception summary	$OP_1 > OP_2$	$OP_1 > OP_2$
2	Result = 0	Invalid operation exception summary	$OP_1 = OP_2$	$OP_1 = OP_2$
3	Summary overflow	Overflow exception	Summary overflow	Unordered (one operand is a Nan)

*RC = Record bit

Table 6.8.3 Interpretation of bits in condition register

6.8.6 Power Interrupts

The Table 6.8.4 shows the various types of interrupts recognized by PowerPC 620. Some interrupts are caused by instruction execution and some are not.

Entry point	Interrupt type	Interrupt caused by instruction execution	Description
00000h	Reserved	Yes	
00100h	System reset	No	Assertion of the processor's hand or soft reset input signals by external logic.
00200h	Machine check	No	Assertion of $\overline{TEA}$ to the processor when it is enabled to recognize machine checks.
00300h	Data storage	Yes	Examples : data page fault; access rights violation on load/store.
00400h	Instruction storage	Yes	Code page fault; attempted instruction fetch from I/O segment; access rights violation
00500h	External	No	Assertion of the processor's external interrupt input signal by external logic when external interrupt recognition is enabled.

00600h	Alignment	Yes	Unsuccessful attempt to access memory due to misaligned operand.
00700h	Program	Yes	Floating-point interrupt; user attempts to execute privileged instruction; trap instruction executed with specified condition met; illegal instruction.
00800h	Floating-point unavailable	Yes	Attempt to execute floating-point instruction with floating-point unit disabled.
00900h	Decrementer	No	Exhaustion of the decrementer register when external interrupt recognition is enabled.
00A00h	Reserved	Yes	
00B00h	Reserved	Yes	
00C00h	System call	Yes	Execution of a system call instruction.
00D00h	Trace	Yes	Single-step or branch trace interrupt.
00E00h	Floating-point assist	Yes	Attempts to execute relatively infrequent, complex floating-point operation (e.g., operation on denormalized number.)
00E10h through 00FFFh	Reserved	Yes	
01000h through 02FFFh	Reserved (implementation specific)	Yes	

Table 6.8.4

6.8.7 Machine Status Register (MSR)

When program is interrupted, it is necessary to store the current state of the processor or machine. So that after interrupt is over it is possible to recover the state of the processor at the time of the interrupt. The state of the processor not only includes the contents of the register but also the contents of various control conditions relating to execution. The machine state register is used to store the state of the processor. The bit pattern for machine status register is shown in the Table 6.8.5.

Bit	Definitions
0	Processor Mode : 32 bit/64-bit
1-44	Reserved
45	Power management : Enabled/Disabled
46	Implementation dependent
47	Interrupt Handler Run Mode : big-endian or little endian
48	External interrupt : Enabled/Disabled
49	Privileged state : Yes/No
50	Availability of floating point unit : Yes/No
51	Machine check interrupt : Enable/Disabled
52	Floating point exception Mode 0
53	Single step trace : Enabled/Disabled
54	Branch trace : Enabled/Disabled
55	Floating point exception Mode 1
56	Reserved
57	Most significant part of exception address 000H/FFFH
58	Instruction address translation ON/OFF
59	Data address translation ON/OFF
60 : 61	Reserved
62	Interrupt recoverable : Yes/No
63	Processor Mode : big-endian/little-endian

Table 6.8.5 PowerPC machine state register

Bits 52 and 55 which represent floating point exception modes are interpreted as given in Table 6.8.6.

FE0	FE1	Mode
0	0	Floating point exceptions disabled
0	1	Floating pont imprecise nonrecoverable
1	0	Floating point imprecise recoverable
1	1	Floating point precise mode

Table 6.8.6 Floating point exception modes

6.8.8 Data Types

The PowerPC architectures recognize the following integer data types :

- Byte : 8-bits (1 byte)
- Halfword : 16-bits (2 bytes)
- Word : 32-bits (4 bytes)
- Doubleword : 64-bits (8 bytes)

It supports both big-endian and little-endian byte ordering. In little-endian, the LSB has the lower address, i.e. the bytes are addressed 3, 2, 1, 0 from the MSB down. On the other hand, in big-endian, the LSB has the upper address, i.e. the bytes are addressed 0, 1, 2, 3 from the MSB down.

PowerPC architecture implements the IEEE 754-1985 standard single and double precision floating point formats.

6.8.9 PowerPC Addressing Modes

In PowerPC, all operations are register-to-register. They use two addressing modes :

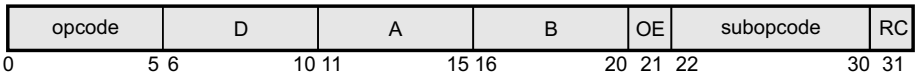
- **Register direct :** In this addressing mode, the operand is either in a general purpose register (GPR) or in floating point register (FPR).
- **Immediate :** In this addressing mode, the operand is a part of the instruction.

For load, store and branch instructions an effective address (EA) to memory is needed. For this following addressing modes are used.

- **Register indirect :** In this addressing mode, a GPR register contains the address of the operand in memory.
- **Register indirect with immediate index :** In this addressing mode, an immediate displacement d is added to the contents of GPR to form an effective address.
- **Register indirect with index :** In this addressing mode, the contents of two GPRs, one serving as register indirect and other as index, is added to form the effective address.
- **Immediate addressing :** In this addressing mode branch address is a part of the instruction. It is sign-extended to form a target effective address.
- **Link register indirect :** In this addressing mode, the link register stores the target effective address.
- **Count register indirect :** In this addressing mode, the count register stores the target effective address.

6.8.10 PowerPC Instruction Formats

The Fig. 6.8.7 shows the main instruction formats for PowerPC. As shown in the Fig. 6.8.7, PowerPC instructions are 32 bits long. For all instructions, the first six bits (Bits 0 to 5) contain the opcode. For some instructions extended opcode is required. Bit 22 to 30 are used for specifying extended opcode. The two source operands A and B are specified by fields A (Bits 11 to 15) and B (Bits 16 to 20) respectively. The destination operand is specified by field D (Bits 6 to 10). The control bit OE (Bit 21) enables overflow detection when set (OE = 1). Another control bit RC (Bit 31), called the record bit, updates CR if set (RC = 1).



OE = extended arithmetic bit
RC = record bit

(a) Register-to-register operations



(b) Immediate operations, load/store register indirect and immediate



AA = absolute address bit
LK = link bit
LI = immediate address field

(c) Branch



(d) Branch conditional



(e) Combined arithmetic (floating-point multiply and add)



(f) Load/store register indirect and index

Fig. 6.8.7 PowerPC main instruction formats

6.8.11 PowerPC Instruction Set

The PowerPC instructions can be categorized in following instruction types :

- Integer arithmetic
- Logical and shift
- Floating-point
- Load/store
- Flow control
- Memory control
- Processor control
- External control

1. Integer arithmetic

Mnemonic	Operation
add	add
addc	add carrying
adde	add extended
addi	add immediate
addic	add immediate carrying
addic.	add immediate carrying and record
addis	add immediate shifted
addme	add to minus one extended
addze	add to zero extended
divs	divide short
divw	divide word
divwu	divide word unsigned
mulhw	multiply high word
mulhwu	multiply high word unsigned
mullw	multiply low
mulli	multiply low immediate
subf	subtract from
subfc	subtract from carrying
subfe	subtract from extended
subfic	subtract from immediate carrying
subfme	subtract from minus one extended
subfze	subtract from zero extended

2. Logical and shift

and	and
andc	and with complement
andi.	immediate
andis	and immediate shifted
cmp	compare
cmpi	compare immediate
cmpl	compare logical
cmpli	compare logical immediate
cntlzw	count leading zeros word
crand	CR AND
crandc	CR AND with complement
creqv	CR equivalent
crnand	CR NAND
crnor	CR NOR
cror	CR OR
crorc	CR OR with complement
crxor	CR XOR
eqv	equivalent
extsb	extend sign byte
extsh	extend sign halfword
isync	instruction synchronize
lbz	load byte and zero
lbzu	load byte and zero with update
lbzux	load byte and zero with update indexed
lbzx	load byte and zero indexed
lfd	load floating-point double-precision
lfdu	load floating-point double-precision with update
lfdux	load floating-point double-precision with update indexed
lfdx	load floating-point double-precision indexed
lfs	load floating-point single-precision
lfsu	load floating-point single-precision with update
lfsux	load floating-point single-precision with update indexed
lfsx	load floating-point single-precision indexed
lha	load halfword algebraic
lhau	load halfword algebraic with update
lhaux	load halfword algebraic with update indexed
lhax	load halfword algebraic indexed
lnbrx	load halfword byte-reverse indexed

lhz	load halfword and zero
lhzu	load half-word and zero with update
lhzux	load halfword and zero with update indexed
lhzx	load halfword and zero indexed
lmw	load multiple word
lswi	load string word immediate
lswx	load string word indexed
lwarx	load word and reverse indexed
lwbrx	load word byte-reverse indexed
lwz	load word and zero
lwzu	load word and zero with update
lwzux	load word and zero with update indexed
lwzx	load word and zero indexed
nand	NAND
neg	negate
nor	NOR
or	OR
orc	OR with complement
ori	OR immediate
oris	OR immediate shifted
rlwimi	rotate left word immediate then mask insert
rlwinm	rotate left word immediate then AND with mask
rlwnm	rotate left word then AND with mask
slw	shift left word
sraw	shift right algebraic word
srawi	shift right algebraic word immediate
srw	shift right word
xor	XOR
xori	XOR immediate
xoris	XOR immediate shifted

3. Floating-point

fabs	floating-point absolute value
fadd	floating-point add
fcmpo	floating-point compare ordered
fcmpu	floating-point compare unordered
fctiw	floating-point convert to integer word
fctiwz	floating-point convert to integer word with round to zero.
fdiv	floating-point divide (single-precision)

fmadd	floating-point multiply-add (single-precision)
fmr	floating-point move register
fmsub	floating-point multiply-subtract (single-precision)
fmul	floating-point multiply (single-precision)
fnabs	floating-point negative absolute value
fneg	floating-point negate
fnmadd	floating-point negative multiply-add (single precision)
fnmsub	floating-point neg. multiply-subtract (single-precision)
frsp	floating-point round to single-precision
fsub	floating-point subtract (single-precision)
mcrfs	move to CR from FPSCR
mffs	move from FPSCR
mtfsb0	move to FPSCR bit 0
mtfsb1	move to FPSCR bit 1
mtfsf	move to FPSCR fields
mtfsfi	move to FPSCR field immediate

4. Load / store

eieio	enforce in-order execution of I/O
stb	store byte
stbu	store byte with update
stbux	store byte with update indexed
stbx	store byte indexed
stfd	store floating-point double-precision
stfdu	store floating-point double-precision with update
stfdux	store floating-point double-precision with update indexed
stfdx	store floating-point double-precision indexed
stfs	store floating-point single-precision
stfsu	store floating-point single-precision with update
stfsux	store floating-point single-precision with update indexed
stfsx	store floating-point single-precision indexed
sth	store halfword
sthbrx	store halfword byte-reverse indexed
sthu	store halfword with update
sthux	store halfword with update indexed
sthx	store halfword indexed
stmw	store multiple word
stswi	store string word immediate
stswx	store string word indexed

stw	store word
stwbrx	store word byte-reverse indexed
stwcx.	store word conditional indexed (CR0 field affected)
stwu	store word with update
stwux	store word with update indexed
stwx	store word indexed
sync	synchronize

5. Flow control

b	branch
bc	branch conditional
bcctr	branch conditional to CRT
bclr	branch conditional to LR
mcrf	move CR field
rfi	return from interrupt
sc	system call
tw	trap word
twi	trap word immediate

6. Memory control

dcbf	data cache block flush
dcbi	data cache block invalidate
dcbst	data cache block store
dcbt	data cache block touch
dcbtst	data cache block touch for store
dcbz	data cache block set to zero
icbi	instruction cache block invalidate
mfsr	move from SR
mfsrin	move from SR indirect
mtsr	move to SR
mtsrin	move to SR indirect
tlbie	TLB invalidate entry

7. Processor control

mcrxr	move to CR from XER
mfcr	move from CR
mfmsr	move from MSR
mfspr	move from SPR
mtcrf	move to CR fields
mtmsr	move to MSR
mtspr	move to SPR

8. External control

eciwx	external control input word indexed
ecowx	external control output word indexed

6.8.12 Features of PowerPC Architecture

The PowerPC architecture defines the following features :

- Separate 32-entry register files for integer and floating-point instructions. The general-purpose registers (GPRs) hold source data for integer arithmetic instructions and the floating-point registers (FPRs) hold source and target data for floating-point arithmetic instructions.
- Instructions for loading and storing data between the memory system and either the FPRs or GPRs.
- Uniform-length instructions to allow simplified instruction pipelining and parallel processing instruction dispatch mechanisms.
- Nondestructive use of registers for arithmetic instructions in which the second, third and sometimes the fourth operand, typically specify source registers for calculations whose results are typically stored in the target register specified by the first operand.
- A precise exception model (with the option of treating floating-point exceptions imprecisely)
- Floating-point support that includes IEEE-754 floating-point operations.
- A flexible architecture definition the allows certain features to be performed in either hardware or with assistance from implementation-specific software depending on the needs of the processor design.
- The ability to perform both single-and double-precision floating-point operations.
- User-level instructions for explicitly storing, flushing and invalidating data in the on-chip caches. The architecture also defines special instructions (cache block touch instructions) for speculatively loading data before it is needed, reducing the effect of memory latency. Definition of a memory model that allows weakly-ordered memory accesses. This allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.
- Support for separate instruction and data caches (Harvard architecture) and for unified caches.
- Support for both big- and little-endian addressing modes.
- Support for 64-bit addressing. The architecture supports both 32-bit or 64-bit implementations.

6.8.13 Features Not Defined by the PowerPC Architecture

Because flexibility is an important design goal of the PowerPC architecture, there are many aspects of the processor design, typically relating to the hardware implementation, that the PowerPC architecture does not define, such as the following:

- **System bus interface signals** - Although numerous implementations may have similar interfaces, the PowerPC architecture does not define individual signals or the bus protocol. For example, the OEA allows each implementation to determine the signal or signals that trigger the machine check exception.
- **Cache design** - The PowerPC architecture does not define the size, structure, the replacement algorithm, or the mechanism used for maintaining cache coherency. The PowerPC architecture supports, but does not require, the use of separate instruction and data caches. Likewise, the PowerPC architecture does not specify the method by which cache coherency is ensured.
- **The number and the nature of execution units** - The PowerPC architecture is RISC architecture, and as such has been designed to facilitate the design of processors that use pipelining and parallel execution units to maximize instruction throughput. However, the PowerPC architecture does not define the internal hardware details of implementations. For example, one processor may execute load and store operations in the integer unit, while another may execute these instructions in a dedicated load/store unit.

Other internal micro-architecture issues - The PowerPC architecture does not prescribe which execution unit is responsible for executing a particular instruction; it also does not define details regarding the instruction fetching mechanism, how instructions are decoded and dispatched, and how results are written back. Dispatch and write-back may occur in order or out of order. Also while the architecture specifies certain registers, such as the GPRs and FPRs, implementations can implement register renaming or other schemes to reduce the impact of data dependencies and register contention.

Review Questions

1. *What is superscalar processor ?*
2. *Explain different instruction issue policies .*
3. *What is register renaming ?*
4. *Explain in detail about super scalar architecture.* **May-11, Marks 10**

6.9 Vector Processor

Dec.-10, Marks 16

A **vector processor,** or **array processor,** is a CPU design where the instruction set includes operations that can perform mathematical operations on multiple data elements simultaneously. This is in contrast to scalar processor which handles one element at a time using multiple instructions. The vast majority of CPUs are scalar (or close to it). Vector processors were common in the scientific computing area, where they formed the basis of most supercomputers.

The need to increase computational power is a never ending requirement. In scientific and research areas, the computations involved are quite extensive and hence high power computers are a must.

The areas like structural engineering, petroleum exploration, aerodynamics, hydrodynamics, nuclear research, tomograph, VLSI design, AI can have data in the form of matrices which suits vector processors to process it at a high speed.

Some examples of it are :

1. In radar and signal processing for detection of space/underwater targets.
2. In remote sensing for earth resources exploration.
3. In computational wind tunnel experiments.
4. In 3D stop action computer assisted tomography.
5. Weather forecasting.
6. Medical diagnosis.

6.9.1 Characteristics of Vector Processing

1. A vector is an ordered set of elements. A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be floating point number, an integer, a logical value, or a character (byte).
2. In vector processing, two successive pairs of elements are processed each clock period. The dual vector pipes and the dual sets of vector functional units allow two pairs of elements to be processed during the same clock period. As each pair of operations is completed, the results are delivered to appropriate elements of the result register. The operation continues until the number of elements processed is equal to the count specified by the vector length register.

For example :

$$C(1 : 50) = A(1 : 50) + B(1 : 50)$$

This vector instruction includes the initial addresses of the two source operands, one destination operand, the length of the vectors and the operation to be performed.

3. Vector instructions are classified into four basic types :

   $f_1 : V \rightarrow V$ $f_2 : V \rightarrow S$

   $f_3 : V \times V \rightarrow V$ $f_4 : V \times S \rightarrow V$

   where V indicates vector operand and S indicates scalar operand. The operations f_1 and f_2 are unary operations such as vector square root, vector sine, vector complement, vector summation and so on. On the other hand, operations f_3 and f_4 are binary operations such as vector add, vector multiply, vector-scalar add and so on.

4. In vector processing, identical processes are repeatedly invoked many times, each of which can be subdivided into subprocesses.
5. In vector processing, successive operands are fed through the pipeline segments and require as few buffers and local controls as possible. This parallel vector processing allow the generation of more than two results per clock period. The parallel vector operations are automatically initiated either when successive vector instructions use different functional units and different vector registers, or when successive vector instructions use the result stream from one vector register as the operand of another operation using a different functional units. This process is known as **chaining**.
6. Because of the startup delay in a pipeline, a vector processors perform better with longer vectors.
7. Vector processing is usually faster and more efficient than scalar processing because it reduces the overhead associated with maintenance of the loop-control variables.

6.9.2 Vector Processing Approach

In order to reduce the amount of time required to access data from memory, most modern CPUs use a technique known as instruction pipelining in which the instructions pass through several sub-units in turn. The first sub-unit reads the address and decodes it, the next "fetches" the values at those addresses, and the next does the math itself. With pipelining the "trick" is to start decoding the next instruction even before the first has left the CPU, in the fashion of an assembly line so the address decoder is constantly in use. Any particular instruction takes the same amount of time of complete, a time known as the latency, but the CPU can process an entire batch of operations much faster than if it did so one at a time.

Vector processors take this concept one step further. Instead of pipelining just the instructions, they also pipeline the data itself. They are fed instructions that say not just to add A to B, but to add all of the numbers "in the array A to all of the numbers from

array B". Instead of constantly having to decode instructions and then fetch the data needed to complete them, it reads a single instruction from memory, and "knows" that the next address will be one larger than the last. This allows for significant savings in decoding time.

To illustrate what a difference this can make, consider the simple task of adding two groups of 10 numbers together. In a normal programming language you would write a "loop" that picked up each of the pairs of numbers in turn, and then added them. To the CPU, this would look something like this :

```
Execute this loop 10 times
read the next instruction and decode it
fetch first number
fetch second number
add them
put the result here
end loop
```

But to vector processor, this task looks considerably different :

```
read instruction and decode it
fetch 10 numbers
fetch 10 numbers
add them
put the results here
```

There are several savings inherent in this approach. For one, only two address translations are needed. Depending on the architecture, this can represent a significant savings by itself. Another savings is fetching and decoding the instruction itself, which only has to be done one time instead of ten. The code itself is also smaller, which can lead to more efficient memory use.

But more than that, the vector processor typically has some form of superscalar implementation, meaning there is not one part of the CPU adding up those 10 numbers, but perhaps two or four of them. Since the output of a vector command does not rely on the input from any other, those two (for instance) parts can each add five of the numbers, thereby completing the whole operation in half the time. This is parallel processing approach for vector computation.

In the above discussion we have seen the vector computation approach. Let us see the processor organization that can be used to implement vector computation approach. The processor organizations used for vector computation can be categorized as :

- Pipelined ALU
- Parallel ALU
- Parallel Processors

Pipelined ALU

The Fig. 6.9.1 (a) shows the pipelined ALU organization. The complex ALU operations such as floating point operations are decomposed into stages. These stages can be pipelined using pipelined ALU. A pipelined ALU will save time only if it is fed a stream of data from sequential locations. A single, isolated floating point operation is not speeded up by a pipeline. The speedup is achieved when a vector of operands is presented to the ALU.

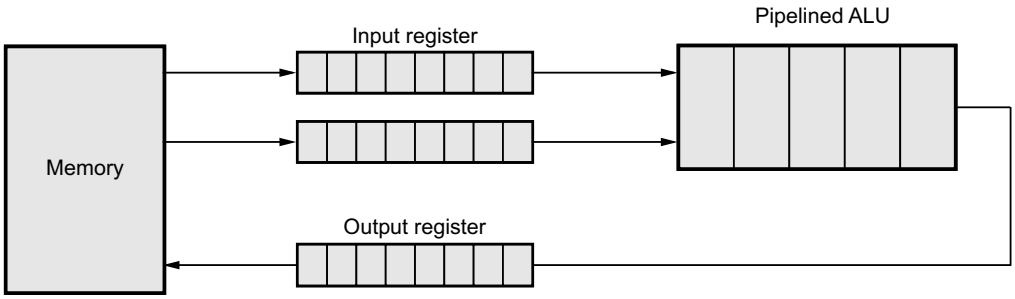


Fig. 6.9.1 (a) Pipelined ALU

Parallel ALUs

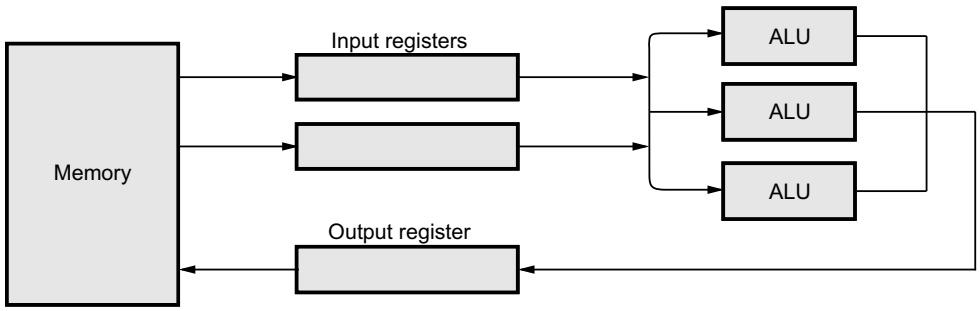


Fig. 6.9.1 (b) Parallel ALUs

The Fig. 6.9.1 (b) shows the parallel ALU organization. Here, multiple ALUs in a single processor are used to implement vector processing approach. In this case, the control unit of the processor routes data to ALUs so that they can function in parallel. It is also possible to use pipelining on each of the parallel ALUs to further speedup the operation.

Parallel Processors

In the parallel processor organization, the multiple ALUs from multiple processors operate in parallel. The ALUs sometimes called **processing units (PE)** are synchronized to perform the same function at the same time. Such processors are also known as **array processors.** In short, an array processor is a synchronous parallel computer with multiple ALUs. The array processor can handle single instruction and multiple data streams. Because of this reason, array processors are also known as SIMD processors.

The Fig. 6.9.2 shows the typical functional structure of an SIMD array processor. As shown in the Fig. 6.9.2, all PEs are synchronized and controlled by single control unit. Each PE is an ALU with attached working registers and local memory. The PEs are interconnected by a data-routing network. The control unit (CU) consists of control processor (CP) and control memory (CM). The system and user programs are executed under control of CU. The user programs are loaded into CU memory from an external source.

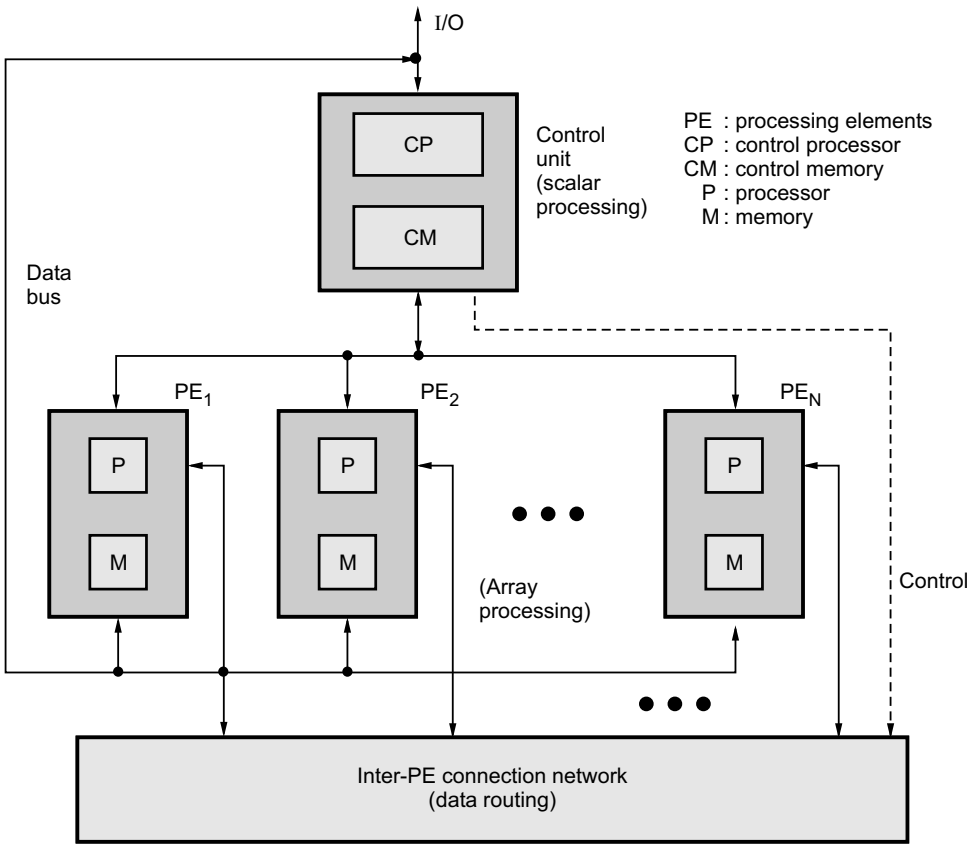


Fig. 6.9.2 Functional structure of SIMD array processor

The function of CU is to decode all the instructions and determine where the decoded instructions should be executed.

- **Scalar instructions** - executed directly inside the CU
- **Vector instructions** - they are broadcast to PEs for through duplicate arithmetic units (PEs)

All the PEs perform the same function synchronously in a lock step fashion under the command of CU. Vector operands are distributed to PE memories before parallel execution in the array of PEs. The distributed data can be loaded into PE memories from an external source via system data bus or via CU in a broadcast mode using control bus. Masking schemes are used to control the status of each PE during execution of vector instruction. A masking vector is used to control the status of all PEs. In other words not all PEs need to participate in the execution of vector instruction.

To enhance the performance of conventional computer and to provide vector processing capability for complex scientific applications, the array processor can be attached to a conventional computer as a peripheral. Due to such attachment the performance of conventional computer improves a lot because of parallel processing support given by array processor with multiple functional units.

The Fig. 6.9.3 shows the interface between the attached array processor and the conventional computer. The array processor is connected to the conventional computer through an input-output controller. Usually, a local memory is available for array processor. The data for array processor is transferred to local memory from the main memory through high-speed bus.

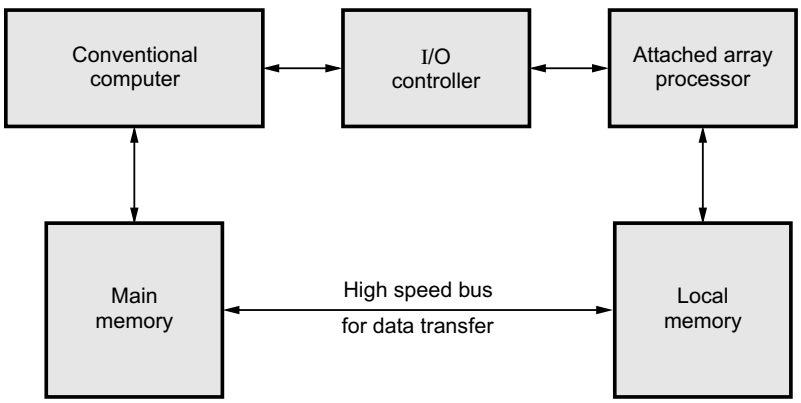


Fig. 6.9.3 Attached array processor with conventional computer

Review Questions

1. *What do you mean by vector computation ?*
2. *State the characteristics of vector processing.*
3. *Explain the following*
   a) Pipelined ALU
   b) Parallel ALU
   c) Parallel Processors
4. *Draw and explain the functional structure of SIMD array processor.*
5. *What are superscalar processors? Explain the typical structure of a typical superscalar processor?*

Dec.-10, Marks 16

Two Marks Questions with Answers

Q.1 ***What is CISC ?***

Ans. : CISC is an acronym for Complex Instruction Set computers or computing. It is based on the concept of using very large instruction set having simple as well as complex instructions and making instruction set more flexible to keep program length as small as possible.

Q.2 ***What is RISC ?***

Ans. : RISC refers to Reduced Instruction Set Computers or Computing. RISC microprocessors are very different from CISC microprocessors. RISC use concept of keeping the instruction set as simple as possible to allow the microprocessor's program to be written using only simple instructions.

Q.3 ***What is the need for reduced instruction chip ?***

Ans. :
- Relatively few instruction types and addressing modes.
- Fixed and easily decoded instruction formats.
- Fast single-cycle instruction execution.
- Hardwired rather than microprogrammed control.

Q.4 ***Compare RISC and CISC architecture.*** **May-10**

Ans. : Refer Table 6.2.1.

Q.5 ***What is superscaler and superscaler processor ?***

Ans. : Processors reach performance levels greater than one instruction per cycle by fetching, decoding and executing several instructions concurrently. This mode of operation is called **superscaler**. A processor capable of parallel instruction execution and

having performance level greater than one instruction per cycle is known as **superscaler processor**.

Q.6 *What is instruction issue degree ?*

Ans. : If the processor can issue up to k instructions simultaneously to the various E-units, then k is called instruction issue degree.

Q.7 *Define parallel processing.*

Ans. : Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

Q.8 *What is register renaming ?*

Ans. : If a temporary register assumes the role of the permanent register whose data it is holding and is given the same name is called the register renaming.

Q.9 *What do you mean by out-of order execution ?*

Ans. : The dispatch unit dispatches the instructions in the order in which they appear in the program. But their execution may be completed in the different order because of data dependency among the instructions. Such situation is called out-of order execution.

Q.10 *What is imprecise and precise exception ?*

Ans. : Situation in which one or more of the succeeding instructions have been executed to completion is called imprecise exception. Situation in which all subsequent instructions that may have been partially executed are discarded. This is called a precise exception.

Q.11 *What is a deadlock ?*

Ans. : A deadlock is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs, which is being held by unit B. at the same time, unit B is waiting for unit A to finish before it can release that resource.

Q.12 *What is the function of commitment unit ?*

Ans. : It is a special control unit needed to guarantee in-order commitment when out-of-order execution is allowed. It uses a special queue called the 'reorder buffer' which stores the instruction(s) committed next.

Q.13 *What is vector processing ?*

Ans. : A vector is a set of scalar data items, all of the same type, stored in memory. The vector processing deals with the arithmetic and logical operations applied to vectors.

Q.14 *What are array processors ?*

Ans. : Processors with multiple arithmetic units, called processing elements (PE), that can operate in parallel and perform computations on large arrays of data. Such processors are known as array processors.

Q.15 *What is an interrupt ?*

Ans. : An interrupt is an event that causes the execution of one program to be suspended and another program to be executed.

Q.16 *What are the uses of interrupts ?*

Ans. :

1. Recovery from errors
2. Debugging
3. Communication between programs
4. Use of interrupts in operating system

Q.17 *What is a bus ?*

Ans. : A collection of wires that connects several devices is called a bus.

Q.18 *What is interconnection structure ?*

Ans. : Computer modules such as central processing unit, memory unit and I/O unit work together with communicating each other and have paths for connecting the modules together. The collection of paths connecting the various modules is called the **interconnection structure**.

Q.19 *What is meant by a multiple bus ? Where it is organised ?*

Dec.-08

Ans. : The need of high speed shared bus is impractical to satisfy with a single bus. Thus, most computer systems use the multiple buses. These buses have the hierarchical structure.

❑❑❑

Notes

Other Popular Books by the Same Authors

1. Advance Processors (8086, 80186/88, 80286, 80386, 80486 and Pentium Processors)
2. Communication Engineering
3. Computer Architecture & Organisation
4. Computer Graphics
5. Computer Graphics & Multimedia
6. Computer Programming using C
7. Computer System and Programming in C
8. Digital Electronics
9. Digital IC Applications
10. Digital Logic Circuits using VHDL
11. Digital Principles and Systems Design using Verilog HDL
12. Electronic Circuits I
13. Electronic Circuits II
14. Electronic Devices and Circuits - I
15. Electronic Devices and Circuits - II
16. Electronics & Microprocessor (Electronic Devices, Introduction to Digital Electronics and 8085 Microprocessor)
17. Embedded Systems
18. Fundamentals of HDL
19. Linear & Digital IC Applications
20. Linear Integrated Circuits & Applications
21. Mechatronics & Microprocessor
22. Microcontrollers (8051 and MSP430 Microcontrollers Family - Architecture & Programming)
23. Microprocessors & Multicore Systems
24. Microprocessor and Control Systems
25. Microprocessor Techniques (8085 and 8086 - Architecture, Instruction set, Programming and Peripheral Interfacing)
26. Microprocessors and Introduction to Microcontroller (8085, 8086, 8051 - Architecture, Interfacing and Programming)
27. Microprocessors and Microcontrollers (8086, 8051 - Architecture, Interfacing and Programming)
28. Microprocessors and Microcontrollers Interfacing Techniques
29. Pulse and Digital Circuits

ISBN 978-93-5099-234-0



TECHNICAL PUBLICATIONS
An Up-Thrust for Knowledge

Amit Residency, Office No.1, 412, Shaniwar Peth, Pune - 411030, M.S. INDIA.
Ph.: +91-020-24495496/97, Telefax : +91-020-24495497
Email : technical@vtubooks.com Website : www.vtubooks.com
eBooks available on www.technicalpublications.org